


# Web Development Essentials

Versión 1.0

Español



# Table of Contents




# Tema 031: Desarrollo de Software y Tecnologías Web




# 031.1 Conceptos básicos de desarrollo de software

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 031.1

**Peso**

1

**Áreas de conocimiento clave**

- Comprender qué es el código fuente
- Comprender los principios de compiladores e intérpretes.
- Comprender el concepto de bibliotecas.
- Comprender los conceptos de programación funcional, procedimental y orientada a objetos.
- Conocimiento de las características comunes de los editores de código fuente y los entornos de desarrollo integrados (IDE)
- Conocimiento de los sistemas de control de versiones.
- Conocimiento de las pruebas de software.
- Conocimiento de lenguajes de programación importantes (C, C++, C#, Java, JavaScript, Python, PHP)




# 031.1 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 031 Desarrollo de software y tecnologías web |
| **Objetivo:** | 031.1 Conceptos básicos de desarrollo de software |
| **Lección:** | 1 de 1 |

## Introducción

Las primeras computadoras se programaron a través del agotador proceso de enchufar cables en enchufes. Los científicos informáticos pronto comenzaron una búsqueda interminable de formas fáciles de decirle a la computadora qué hacer. Este capítulo presenta las herramientas de programación. Analiza las formas clave en que las instrucciones de texto (lenguajes de programación) representan las tareas que un programador desea realizar y las herramientas que transforman el programa en una forma denominada *lenguaje de máquina* que una computadora puede ejecutar.

**NOTE** En este texto, los términos *programa* y *aplicación* se usan indistintamente.

### Código Fuente

Un programador normalmente desarrolla una aplicación escribiendo una descripción textual, llamada *código fuente*, de la tarea deseada. El código fuente está escrito en un *lenguaje de programación* cuidadosamente definido que representa lo que la computadora puede hacer en una abstracción de alto nivel que los humanos pueden entender. También se han desarrollado

herramientas para permitir a los programadores, así como a los no programadores, expresar sus pensamientos visualmente, pero escribir el código fuente sigue siendo la forma predominante de programar.

De la misma manera que un lenguaje natural tiene sustantivos, verbos y construcciones para expresar ideas de manera estructurada, las palabras y la puntuación en un lenguaje de programación son representaciones simbólicas de operaciones que se realizarán en la máquina.

En este sentido, el código fuente no es muy diferente de cualquier otro texto en el que el autor emplea las reglas bien establecidas de un lenguaje natural para comunicarse con el lector. En el caso del código fuente, el "lector" es la máquina, por lo que el texto no puede contener ambigüedades o inconsistencias, ni siquiera sutiles.

Y como cualquier texto que discute algún tema en profundidad, el código fuente también necesita estar bien estructurado y organizado lógicamente al desarrollar aplicaciones complejas. Se pueden almacenar programas muy simples y ejemplos didácticos en unas pocas líneas de un solo archivo de texto, que contiene todo el código fuente del programa. Los programas más complejos se pueden subdividir en miles de archivos, cada uno con miles de líneas.

El código fuente de las aplicaciones profesionales debe organizarse en diferentes carpetas, generalmente asociadas con un propósito particular. Un programa de chat, por ejemplo, se puede organizar en dos carpetas: una que contiene los archivos de código que manejan la transmisión y recepción de mensajes a través de la red, y otra carpeta que contiene los archivos que construyen la interfaz y reaccionan a las acciones del usuario. De hecho, es común tener muchas carpetas y subcarpetas con archivos de código fuente dedicados a tareas muy específicas dentro de la aplicación.

Además, el código fuente no siempre está aislado en sus propios archivos, con todo escrito en un solo lenguaje. En aplicaciones web, por ejemplo, un documento HTML puede incrustar código JavaScript para complementar el documento con funcionalidad adicional.

### Editores de código e IDE

La variedad de formas en que se puede escribir el código fuente puede resultar intimidante. Por lo tanto, muchos desarrolladores aprovechan las herramientas que ayudan a escribir y probar el programa.

El archivo de código fuente es solo un archivo de texto sin formato. Como tal, puede ser editado por cualquier editor de texto, sin importar lo simple que sea. Para que sea más fácil distinguir entre el código fuente y el texto sin formato, cada lenguaje adopta una extensión de nombre de archivo autoexplicativa: `.c` para el lenguaje C, `.py` para Python, `.js` para JavaScript, etc. Los editores de propósito general a menudo entienden el código fuente de los lenguajes populares lo suficientemente bien como para agregar cursiva, colores y sangría para que el código sea comprensible.

No todos los desarrolladores optan por editar el código fuente en un editor de propósito general. Un *entorno de desarrollo integrado* (IDE) proporciona un editor de texto junto con herramientas para ayudar al programador a evitar errores sintácticos e inconsistencias obvias. Estos editores se recomiendan especialmente para programadores menos experimentados, pero los programadores experimentados también los usan.

Los IDE populares como Visual Studio, Eclipse y Xcode observan inteligentemente lo que escribe el programador, sugiriendo con frecuencia palabras para usar (autocompletado) y verificando el código en tiempo real. Los IDE pueden incluso ofrecer pruebas y depuración automatizadas para identificar problemas cada vez que cambia el código fuente.

Algunos programadores más experimentados optan por editores menos intuitivos como Vim, que ofrecen una mayor flexibilidad y no requieren la instalación de paquetes adicionales. Estos programadores usan herramientas externas e independientes para agregar las funciones que están integradas cuando usa un IDE.

### Mantenimiento de código

Ya sea en un IDE o usando herramientas independientes, es importante emplear algún tipo de *sistema de control de versiones* (VCS). El código fuente está en constante evolución porque es necesario corregir fallas imprevistas y deben incorporarse mejoras. Una consecuencia inevitable de esta evolución es que las correcciones y mejoras pueden interferir con otras partes de las aplicaciones en una gran base de código. Las herramientas de control de versiones como Git, Subversion y Mercurial registran todos los cambios realizados en el código y quién realizó el cambio, lo que le permite rastrear y eventualmente recuperarse de una modificación fallida.

Además, las herramientas de control de versiones permiten a cada desarrollador del equipo trabajar en una copia de los archivos de código fuente sin interferir con el trabajo de otros programadores. Una vez que las nuevas versiones del código fuente están listas y probadas, otros miembros del equipo pueden incorporar las correcciones o mejoras realizadas en una copia.

Git, el sistema de control de versiones más popular en la actualidad, permite que diferentes personas mantengan muchas copias independientes de un repositorio, que comparten sus cambios como desean. Sin embargo, ya sea que utilicen un sistema de control de versiones descentralizado o centralizado, la mayoría de los equipos mantienen un repositorio en cuyo código fuente y recursos se puede confiar. Varios servicios en línea ofrecen almacenamiento para repositorios de código fuente. Los más populares de estos servicios son GitHub y GitLab, pero también vale la pena mencionar Savannah del proyecto GNU.

## Lenguajes de programación

Existe una amplia variedad de lenguajes de programación; cada década ve la invención de otros nuevos. Cada lenguaje de programación tiene sus propias reglas y se recomienda para propósitos particulares. Aunque los lenguajes muestran diferencias superficiales en sintaxis y palabras claves, lo que realmente distingue a los lenguajes son los profundos enfoques conceptuales que representan, conocidos como *paradigmas*.

### Paradigmas

Los paradigmas definen las premisas en las que se basa un lenguaje de programación, especialmente en lo que respecta a cómo debe estructurarse el código fuente.

El desarrollador parte del paradigma del lenguaje para formular las tareas a realizar por la máquina. Estas tareas, a su vez, se expresan simbólicamente con las palabras y construcciones sintácticas que ofrece el lenguaje.

El lenguaje de programación es *procedural* cuando las instrucciones presentadas en el código fuente se ejecutan en orden secuencial, como un guión de película. Si el código fuente está segmentado en funciones o subrutinas, una rutina principal se encarga de llamar a las funciones en secuencia.

El siguiente código es un ejemplo de lenguaje de procedimiento. Escrito en C, define variables para representar el lado, área y volumen de formas geográficas. El valor de la variable `side` se asigna en `main()`, que es la función que se invoca cuando se ejecuta el programa. Las variables `area` y `volume` se calculan en las subrutinas `square()` y `cube()` que preceden a la función principal:

```
#include <stdio.h>

float side;
float area;
float volume;

void square(){ area = side * side; }

void cube(){ volume = area * side; }

int main(){
  side = 2;
  square();
  cube();
  printf("Volume: %f\n", volume);
  return 0;
}
```

El orden de acciones definido en `main()` determina la secuencia de estados del programa, caracterizada por el valor de las variables `side`, `area` y `volume`. El ejemplo termina después de mostrar el valor de `volume` con la declaración `printf`.

Por otro lado, el paradigma de la *programación orientada a objetos* (POO) tiene como característica principal la separación del estado del programa en subestados independientes. Estos subestados y operaciones asociadas son los *objetos*, así llamados porque tienen una existencia más o menos independiente dentro del programa y porque tienen propósitos específicos.

Los distintos paradigmas no restringen necesariamente el tipo de tarea que puede realizar un programa. El código del ejemplo anterior se puede reescribir de acuerdo con el paradigma OOP usando el lenguaje C++:

```
#include <iostream>

class Cube {
  float side;
  public:
  Cube(float s){ side = s; }
  float volume() { return side * side * side; }
};

int main(){
  float side = 2;
  Cube cube(side);
  std::cout << "Volume: " << cube.volume() << std::endl;
  return 0;
}
```

La función `main()` todavía está presente. Pero ahora hay una nueva palabra, `class`, que introduce la definición de un objeto. La clase definida, denominada `Cube`, contiene sus propias variables y subrutinas. En OOP, una variable también se llama *attribute* y una subrutina se llama *method*.

Está fuera del alcance de este capítulo explicar todo el código C++ en el ejemplo. Lo que es importante para nosotros aquí es que `Cube` contiene el atributo `side` y dos métodos. El método `volume()` calcula el volumen del cubo.

Es posible crear varios objetos independientes de la misma clase y las clases pueden estar compuestas por otras clases.

Tenga en cuenta que estas mismas características se pueden escribir de manera diferente y que los ejemplos de este capítulo están demasiado simplificados. C y C++ tienen características mucho más sofisticadas que permiten construcciones mucho más complejas y prácticas.

La mayoría de los lenguajes de programación no imponen rigurosamente un paradigma, éstos permiten a los programadores elegir varios aspectos de un paradigma u otro. JavaScript, por ejemplo, incorpora aspectos de diferentes paradigmas. El programador puede descomponer todo el programa en funciones que no comparten un estado común entre sí:

```
function cube(side){
  return side*side*side;
}

console.log("Volume: " + cube(2));
```

Aunque este ejemplo es similar a la programación procedimental, tenga en cuenta que la función recibe una copia de toda la información necesaria para su ejecución y siempre produce el mismo resultado para el mismo parámetro, independientemente de los cambios que ocurran fuera del alcance de la función. Este paradigma, llamado *funcional*, está fuertemente influenciado por el formalismo matemático, donde cada operación es autosuficiente.

Otro paradigma cubre los lenguajes *declarativos*, que describen los estados en los que desea que esté el sistema. Un lenguaje declarativo puede descubrir cómo lograr los estados especificados. SQL, el lenguaje universal para consultar bases de datos, a veces se denomina lenguaje declarativo, aunque realmente ocupa un nicho único en el panteón de la programación.

No existe un paradigma universal que pueda adaptarse a cualquier contexto. La elección del lenguaje también puede estar restringida por los lenguajes admitidos en la plataforma o el entorno de ejecución donde se utilizará el programa.

Una aplicación web que será utilizada por el navegador, por ejemplo, deberá estar escrita en JavaScript, que es un lenguaje compatible universalmente con los navegadores. (Se pueden usar algunos otros lenguajes porque proporcionan convertidores para crear JavaScript). Por lo tanto, para el navegador web, a veces llamado *client side* o *front end* de la aplicación web, el desarrollador tendrá que usar los paradigmas permitidos en JavaScript. El lado del servidor o el back-end de la aplicación, que maneja las solicitudes desde el navegador, normalmente se programa en un lenguaje diferente; PHP es el más popular para este propósito.

Independientemente del paradigma, cada lenguaje tiene *bibliotecas* predefinidas de funciones que se pueden incorporar al código. Las funciones matemáticas, como las que se ilustran en el código de ejemplo, no necesitan implementarse desde cero, ya que el lenguaje ya tiene la función lista para usar. JavaScript, por ejemplo, proporciona el objeto `Math` con las operaciones matemáticas más comunes.

Incluso las funciones más especializadas suelen estar disponibles a través del proveedor del lenguaje o de desarrolladores externos. Estas bibliotecas de recursos adicionales pueden estar en forma de código fuente; es decir, en archivos adicionales que se incorporan al archivo donde se utilizarán. En JavaScript, la incrustación se realiza con `import from`:

```
import { OrbitControls } from 'modules/OrbitControls.js';
```

Este tipo de importación, donde el recurso incrustado es también un archivo de código fuente, se usa con mayor frecuencia en los llamados *lenguajes interpretados*. Los *lenguajes compilados* permiten, entre otras cosas, la incorporación de funcionalidades precompiladas en lenguaje máquina, es decir, bibliotecas *compiladas*. La siguiente sección explica las diferencias entre estos tipos de lenguajes.

### Compiladores e Intérpretes

Como ya sabemos, el código fuente es una representación simbólica de un programa que necesita ser traducido al lenguaje de máquina para poder ejecutarse.

A grandes rasgos, hay dos formas posibles de realizar la traducción: convirtiendo el código fuente de antemano para su ejecución futura o convirtiendo el código en el momento de su ejecución. Los lenguajes de la primera modalidad se denominan *lenguajes compilados* y los lenguajes de la segunda modalidad se denominan *lenguajes interpretados*. Algunos lenguajes interpretados ofrecen la compilación como una opción, para que el programa pueda iniciarse más rápido.

En los lenguajes compilados, existe una clara distinción entre el código fuente del programa y el programa en sí, que será ejecutado por la computadora. Una vez compilado, el programa generalmente funcionará solo en el sistema operativo y la plataforma para los que fue compilado.

En un lenguaje interpretado, el código fuente en sí se trata como el programa y el proceso de conversión a lenguaje de máquina es transparente para el programador. Para un lenguaje interpretado, es común llamar al código fuente un *script*. El intérprete traduce el script al lenguaje de máquina para el sistema en el que se está ejecutando.

## Compilación y compiladores

El lenguaje de programación C es uno de los ejemplos más conocidos de lenguaje compilado. Las mayores fortalezas del lenguaje C son su flexibilidad y rendimiento. Tanto las supercomputadoras de alto rendimiento como los microcontroladores de los electrodomésticos se pueden programar en lenguaje C. Otros ejemplos de lenguajes compilados populares son C++ y C# (C sharp). Como sugieren sus nombres, estos lenguajes están inspirados en C, pero incluyen características que admiten el paradigma orientado a objetos.

El mismo programa escrito en C o C++ se puede compilar para diferentes plataformas, requiriendo poco o ningún cambio en el código fuente. Es el compilador el que define la plataforma de destino del programa. Hay compiladores específicos de plataforma, así como compiladores multiplataforma como GCC (que significa *GNU Compiler Collection*) que pueden producir programas binarios para muchas arquitecturas distintas.

**NOTE** También existen herramientas que automatizan el proceso de compilación. En lugar de invocar al compilador directamente, el programador crea un archivo que indica los diferentes pasos de compilación que se realizarán automáticamente. La herramienta tradicional utilizada para este propósito es `make`, pero también se utilizan de forma generalizada varias herramientas más nuevas, como Maven y Gradle. Todo el proceso de construcción está automatizado cuando se usa un IDE.

El proceso de compilación no siempre genera un programa binario en lenguaje de máquina. Hay lenguajes compilados que producen programas en un formato genéricamente llamado *bytecode*. Al igual que un script, el código de bytes no está en un lenguaje específico de la plataforma, por lo que requiere un programa de interpretación que lo traduzca al lenguaje de máquina. En este caso, el programa de interpretación simplemente se denomina *runtime*.

El lenguaje Java adopta este enfoque, por lo que los programas compilados escritos en Java se pueden utilizar en diferentes sistemas operativos. A pesar de su nombre, Java no está relacionado con JavaScript.

El código de bytes está más cerca del lenguaje de máquina que el código fuente, por lo que su ejecución tiende a ser comparativamente más rápida. Debido a que todavía hay un proceso de conversión durante la ejecución del código de bytes, es difícil obtener el mismo rendimiento que un programa equivalente compilado en lenguaje de máquina.

## Interpretación e intérpretes

En lenguajes interpretados como JavaScript, Python y PHP, no es necesario precompilar el programa, lo que facilita su desarrollo y modificación. En lugar de compilarlo, el script lo ejecuta otro programa llamado intérprete. Por lo general, el intérprete de un lenguaje recibe el nombre del lenguaje en sí. El intérprete de una secuencia de comandos de Python, por ejemplo, es un programa llamado `python`. El intérprete de JavaScript suele ser el navegador web, pero el programa `node` también puede ejecutar scripts fuera de un navegador. Debido a que se convierte en instrucciones binarias cada vez que se ejecuta, un programa en lenguaje interpretado tiende a ser más lento que un equivalente en lenguaje compilado.

Nada impide que la misma aplicación tenga componentes escritos en diferentes idiomas. Si es necesario, estos componentes pueden comunicarse a través de una *interfaz de programación de aplicaciones* (API) mutuamente comprensible.

El lenguaje Python, por ejemplo, tiene capacidades de tabulación y minería de datos muy sofisticadas. El desarrollador puede elegir Python para escribir las partes del programa que tratan estos aspectos y otro lenguaje, como C++, para realizar el procesamiento numérico más pesado. Es posible adoptar esta estrategia incluso cuando no existe una API que permita la comunicación directa entre los dos componentes. El código escrito en Python puede generar un archivo en el formato adecuado para ser utilizado por un programa escrito en C++, por ejemplo.

Aunque es posible escribir casi cualquier programa en cualquier lenguaje, el desarrollador debe adoptar el que más se ajuste al propósito de la aplicación. Al hacerlo, se beneficia de la reutilización de componentes ya probados y bien documentados.

## Ejercicios guiados

1. ¿Qué tipo de programa se puede utilizar para editar el código fuente?

2. ¿Qué tipo de herramienta ayuda a integrar el trabajo de diferentes desarrolladores en la misma base de código?

## Ejercicios Exploratorios

1. Suponga que desea escribir un juego en 3D para jugarlo en el navegador. Las aplicaciones web y los juegos están programados en JavaScript. Aunque es posible escribir todas las funciones gráficas desde cero, es más productivo utilizar una biblioteca preparada para este propósito. ¿Qué bibliotecas de terceros proporcionan capacidades para la animación 3D en JavaScript?

2. Además de PHP, ¿qué otros lenguajes se pueden utilizar en el lado del servidor de una aplicación web?

## Resumen

Esta lección cubre los conceptos más esenciales del desarrollo de software. El desarrollador debe conocer los lenguajes de programación importantes y el escenario de uso adecuado para cada uno. Esta lección abarca los siguientes conceptos y procedimientos:

- Qué es el código fuente.
- Editores de código fuente y herramientas relacionadas.
- Paradigmas de programación procedimental, orientada a objetos, funcional y declarativa.
- Características de los lenguajes compilados e interpretados.

## Respuestas a los ejercicios guiados

1. ¿Qué tipo de programa se puede utilizar para editar el código fuente?

   En principio, cualquier programa capaz de editar texto plano.

2. ¿Qué tipo de herramienta ayuda a integrar el trabajo de diferentes desarrolladores en la misma base de código?

   Un sistema de control de versiones o fuentes, como Git.

## Respuestas a los ejercicios de exploración

1. Suponga que desea escribir un juego en 3D para jugarlo en el navegador. Las aplicaciones web y los juegos están programados en JavaScript. Aunque es posible escribir todas las funciones gráficas desde cero, es más productivo utilizar una biblioteca preparada para este propósito. ¿Qué bibliotecas de terceros proporcionan capacidades para la animación 3D en JavaScript?

   Hay muchas opciones para bibliotecas de gráficos 3D para JavaScript, como threejs y BabylonJS.

2. Además de PHP, ¿qué otros lenguajes se pueden utilizar en el lado del servidor de una aplicación web?

   Cualquier idioma admitido por la aplicación del servidor HTTP utilizada en el host del servidor. Algunos ejemplos son Python, Ruby, Perl y el propio JavaScript.




# 031.2 Arquitectura de aplicaciones web

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 031.2

**Peso**

2

**Áreas de conocimiento clave**

- Comprender el principio de la informática de cliente y servidor.
- Comprender el papel de los navegadores web y ser consciente de los navegadores web de uso común
- Comprender el papel de los servidores web y los servidores de aplicaciones.
- Comprender las tecnologías y estándares comunes de desarrollo web.
- Comprender los principios de las API
- Comprender el principio de las bases de datos relacionales y no relacionales (NoSQL)
- Conocimiento de los sistemas de gestión de bases de datos de código abierto de uso común.
- Conocimiento de REST y GraphQL
- Conocimiento de las aplicaciones de una sola página
- Conocimiento del empaquetado de aplicaciones web
- Conocimiento de WebAssembly
- Conocimiento de los sistemas de gestión de contenido.

### Lista parcial de archivos, términos y utilidades

- Chrome, Edge, Firefox, Safari, Internet Explorer
- HTML, CSS, JavaScript
- SQLite, MySQL, MariaDB, PostgreSQL
- MongoDB, CouchDB, Redis




## 031.2 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 031 Desarrollo de software y tecnologías web |
| **Objetivo:** | 031.2 Arquitectura de aplicaciones web |
| **Lección:** | 1 de 1 |

# Introducción

La palabra *aplicación* tiene un significado amplio en la jerga tecnológica. Cuando la aplicación es un programa tradicional, ejecutado localmente y autosuficiente en su propósito, tanto la interfaz operativa de la aplicación como los componentes de procesamiento de datos se integran en un solo “paquete”. Una *aplicación web* es diferente porque adopta el modelo cliente/servidor y su parte cliente se basa en HTML, que se obtiene del servidor y, en general, se procesa mediante un navegador.

## Clientes y servidores

En el modelo cliente/servidor, parte del trabajo se realiza localmente en el *lado del cliente* y parte del trabajo se realiza de forma remota, en el *lado del servidor*. Las tareas que realiza cada parte varían de acuerdo con el propósito de la aplicación, pero en general depende del cliente proporcionar una interfaz al usuario y diseñar el contenido de una manera atractiva. Depende del servidor ejecutar la

aplicación, procesando y respondiendo a las solicitudes realizadas por el cliente. En una aplicación de compras, por ejemplo, la aplicación cliente muestra una interfaz para que el usuario elija y pague los productos, pero la fuente de datos y los registros de transacciones se mantienen en el servidor remoto, al que se accede a través de la red. Las aplicaciones web realizan esta comunicación a través de Internet, generalmente a través del Protocolo de transferencia de hipertexto (HTTP).

Una vez cargada por el navegador, el lado del cliente de la aplicación inicia la interacción con el servidor siempre que sea necesario o conveniente. Los servidores de aplicaciones web ofrecen una *interfaz de programación de aplicaciones* (API) que define las solicitudes disponibles y cómo se deben realizar. Por lo tanto, el cliente construye una solicitud en el formato definido por la API y la envía al servidor, que verifica los requisitos previos para la solicitud y devuelve la respuesta adecuada.

Si bien el cliente, en forma de aplicación móvil o navegador de escritorio, es un programa autónomo con respecto a la interfaz de usuario y las instrucciones para comunicarse con el servidor, el navegador debe obtener la página HTML y los componentes asociados, como imágenes, CSS y JavaScript: que definen la interfaz y las instrucciones para comunicarse con el servidor.

Los lenguajes de programación y las plataformas que utilizan el cliente y el servidor son independientes, pero utilizan un protocolo de comunicación mutuamente comprensible. La parte del servidor casi siempre la realiza un programa sin una interfaz gráfica, que se ejecuta en entornos informáticos de alta disponibilidad para que siempre esté listo para responder a las solicitudes. Por el contrario, la parte del cliente se ejecuta en cualquier dispositivo que sea capaz de representar una interfaz HTML, como los teléfonos inteligentes.

Además de ser imprescindible para determinados fines, la adopción del modelo cliente/servidor permite que una aplicación optimice varios aspectos de desarrollo y mantenimiento, ya que cada parte puede diseñarse para su propósito específico. Una aplicación que muestra mapas y rutas, por ejemplo, no necesita tener todos los mapas almacenados localmente. Solo se requieren mapas relacionados con la ubicación de interés de los usuarios, por lo que solo esos mapas se solicitan al servidor central.

Los desarrolladores tienen control directo sobre el servidor, por lo que también pueden modificar el cliente que les proporciona. Esto permite a los desarrolladores mejorar la aplicación, en mayor o menor medida, sin necesidad de que el usuario instale explícitamente nuevas versiones.

## El lado del cliente

Una aplicación web debe ejecutarse de la misma manera en cualquiera de los navegadores más populares, siempre que el navegador esté actualizado. Algunos navegadores pueden ser incompatibles con innovaciones recientes, pero solo las aplicaciones experimentales utilizan características que aún no se han adoptado ampliamente.

Los problemas de incompatibilidad eran más comunes en el pasado, cuando los diferentes navegadores tenían su propio *motor de renderizado* y había menos cooperación en la formulación y adopción de estándares. El motor de renderizado es el componente principal del navegador, ya que es responsable de transformar HTML y otros componentes asociados en los elementos visuales e interactivos de la interfaz. Algunos navegadores, en particular Internet Explorer, necesitaban un tratamiento especial en el código para no interrumpir el funcionamiento esperado de las páginas.

Hoy en día, existen diferencias mínimas entre los principales navegadores y las incompatibilidades son raras. De hecho, los navegadores Chrome y Edge usan el mismo motor de renderizado (llamado Blink). El navegador Safari y otros navegadores que se ofrecen en la App Store de iOS utilizan el motor WebKit. Firefox usa un motor llamado Gecko. Estos tres motores representan prácticamente todos los navegadores que se utilizan en la actualidad. Aunque se desarrollan por separado, los tres motores son proyectos de código abierto y existe una cooperación entre sus desarrolladores, lo que facilita la compatibilidad, el mantenimiento y la adopción de estándares.

Debido a que los desarrolladores de navegadores se han esforzado mucho para mantener la compatibilidad, el servidor normalmente no está vinculado a un solo tipo de cliente. En principio, un servidor HTTP puede comunicarse con cualquier cliente que también sea capaz de comunicarse a través de HTTP. En una aplicación de mapas, por ejemplo, el cliente puede ser una aplicación móvil o un navegador que carga la interfaz HTML desde el servidor.

## Variedades de clientes web

Hay aplicaciones móviles y de escritorio cuya interfaz se procesa a partir de HTML y, al igual que los navegadores, pueden usar JavaScript como lenguaje de programación. Sin embargo, a diferencia del cliente cargado en el navegador, el HTML y los componentes necesarios para el funcionamiento del cliente nativo están presentes localmente desde la instalación de la aplicación. De hecho, una aplicación que funciona de esta manera es prácticamente idéntica a una página HTML (es probable que ambas sean procesadas por el mismo motor). También existen *aplicaciones web progresivas* (PWA), un mecanismo que le permite empaquetar clientes de aplicaciones web para su uso sin conexión, limitado a funciones que no requieren comunicación inmediata con el servidor. En cuanto a lo que puede hacer la aplicación, no hay diferencia entre ejecutar el navegador o empaquetarla en una PWA, salvo que en esta última el desarrollador tiene más control sobre lo que se almacena localmente.

La representación de interfaces desde HTML es una actividad tan recurrente que el motor suele ser un componente de software independiente, presente en el sistema operativo. Su presencia como componente independiente permite que diferentes aplicaciones lo incorporen sin tener que incrustarlo en el paquete de la aplicación. Este modelo también delega el mantenimiento del motor de renderizado al sistema operativo, facilitando las actualizaciones. Es muy importante mantener actualizado un componente tan crucial para evitar posibles fallas.

Independientemente de su método de entrega, las aplicaciones escritas en HTML se ejecutan en una capa de abstracción creada por el motor, que funciona como un entorno de ejecución aislado. En particular, en el caso de un cliente que se ejecuta en el navegador, la aplicación tiene a su disposición únicamente aquellos recursos que ofrece el navegador. Las funciones básicas, como la interacción con los elementos de la página y la solicitud de archivos a través de HTTP, siempre están disponibles. Los recursos que pueden contener información confidencial, como el acceso a archivos locales, la ubicación geográfica, la cámara y el micrófono, requieren una autorización explícita del usuario antes de que la aplicación pueda usarlos.

## Lenguajes de un cliente web

El elemento central de un cliente de aplicación web que se ejecuta en el servidor es el documento HTML. Además de presentar los elementos de la interfaz que muestra el navegador de forma estructurada, el documento HTML contiene las direcciones de todos los archivos necesarios para la correcta presentación y funcionamiento del cliente.

HTML por sí solo no tiene mucha versatilidad para construir interfaces más elaboradas y no tiene características de programación de propósito general. Por esta razón, un documento HTML que debería funcionar como una aplicación cliente siempre va acompañado de uno o más conjuntos de CSS y JavaScript.

El CSS se puede proporcionar como un archivo separado o directamente en el propio archivo HTML. El propósito principal de CSS es ajustar la apariencia y el diseño de los elementos de la interfaz HTML. Aunque no es estrictamente necesario, las interfaces más sofisticadas suelen requerir modificaciones en las propiedades CSS de los elementos para satisfacer sus necesidades.

JavaScript es un componente prácticamente indispensable. Los procedimientos escritos en JavaScript responden a eventos en el navegador. Estos eventos pueden ser causados por el usuario o no interactivos. Sin JavaScript, un documento HTML está prácticamente limitado a texto e imágenes. El uso de JavaScript en documentos HTML le permite ampliar la interactividad más allá de los hipervínculos y formularios, haciendo que el navegador muestre la página como una interfaz de aplicación convencional.

JavaScript es un lenguaje de programación de propósito general, pero su uso principal es en aplicaciones web. Se puede acceder a las características del entorno de ejecución del navegador a través de palabras claves de JavaScript, que se utilizan en un script para realizar la operación deseada. El término `document`, por ejemplo, se utiliza en el código JavaScript para hacer referencia al documento HTML asociado con el código JavaScript. En el contexto del lenguaje JavaScript, `document` es un *objeto global* con propiedades y métodos que se pueden usar para obtener información de cualquier elemento en el documento HTML. Más importante aún, puede usar el objeto `document` para modificar sus elementos y asociarlos con acciones personalizadas escritas en

JavaScript.

Una aplicación cliente basada en tecnologías web es multiplataforma, porque puede ejecutarse en cualquier dispositivo que tenga un navegador web compatible.

Sin embargo, estar confinado al navegador impone limitaciones a las aplicaciones web en comparación con las aplicaciones nativas. La intermediación realizada por el navegador permite una programación de mayor nivel y aumenta la seguridad, pero también aumenta el procesamiento y el consumo de memoria.

Los desarrolladores trabajan continuamente en los navegadores para proporcionar más funciones y mejorar el rendimiento de las aplicaciones JavaScript, pero existen aspectos intrínsecos a la ejecución de scripts como JavaScript que les imponen una desventaja en comparación con los programas nativos para el mismo hardware.

Una característica que mejora significativamente el rendimiento de las aplicaciones JavaScript que se ejecutan en el navegador es *WebAssembly*. WebAssembly es un tipo de JavaScript compilado que produce código fuente escrito en un lenguaje de bajo nivel más eficiente, como el lenguaje C. WebAssembly puede acelerar principalmente las actividades que requieren un uso intensivo del procesador, ya que evita gran parte de la traducción realizada por el navegador cuando se ejecuta un programa escrito en JavaScript convencional.

Independientemente de los detalles de implementación de la aplicación, todo el código HTML, CSS, JavaScript y archivos multimedia deben obtenerse primero del servidor. El navegador obtiene estos archivos como una página de Internet, es decir, con una dirección a la que accede el navegador.

Una página web que actúa como interfaz para una aplicación web es como un documento HTML simple, pero agrega comportamientos adicionales. En las páginas convencionales, el usuario es dirigido a otra página después de hacer clic en un enlace. Las aplicaciones web pueden presentar su interfaz y responder a los eventos del usuario sin cargar nuevas páginas en la ventana del navegador. La modificación de este comportamiento estándar en páginas HTML se realiza mediante programación JavaScript.

Un cliente de correo web, por ejemplo, muestra mensajes y cambia entre carpetas de mensajes sin salir de la página. Esto es posible porque el cliente usa JavaScript para reaccionar a las acciones del usuario y realizar las solicitudes adecuadas al servidor. Si el usuario hace clic en el asunto de un mensaje en la bandeja de entrada, un código JavaScript asociado con este evento solicita el contenido de ese mensaje del servidor (utilizando la llamada API correspondiente). Tan pronto como el cliente recibe la respuesta, el navegador muestra el mensaje en la parte correspondiente de la misma página. Los diferentes clientes de correo web pueden adoptar diferentes estrategias, pero todos utilizan este mismo principio.

Por tanto, además de proporcionar al navegador los archivos que componen el cliente, el servidor también debe ser capaz de atender solicitudes como la del cliente webmail, cuando solicita el contenido de un mensaje específico. Cada solicitud que puede realizar el cliente tiene un procedimiento predefinido para responder en el servidor, cuya API puede definir diferentes métodos para identificar a qué procedimiento se refiere la solicitud. Los métodos más comunes son:

- Direcciones, a través de un localizador uniforme de recursos (URL)
- Campos en el encabezado HTTP
- Métodos GET/POST
- WebSockets

Un método puede ser más adecuado que otro, dependiendo del propósito de la solicitud y otros criterios que tenga en cuenta el desarrollador. En general, las aplicaciones web utilizan una combinación de métodos, cada uno en una circunstancia específica.

El paradigma *Representational State Transfer* (REST) se usa ampliamente para la comunicación en aplicaciones web, porque se basa en los métodos básicos disponibles en HTTP. El encabezado de una solicitud HTTP comienza con una palabra clave que define la operación básica a realizar: `GET`, `POST`, `PUT`, `DELETE`, etc., acompañada de la URL correspondiente donde se aplicará la acción. Si la aplicación requiere operaciones más específicas, con una descripción más detallada de la operación solicitada, el protocolo GraphQL puede ser una opción más adecuada.

Las aplicaciones desarrolladas utilizando el modelo cliente/servidor están sujetas a inestabilidades en la comunicación. Por ello, la aplicación cliente siempre debe adoptar estrategias de transferencia de datos eficientes para favorecer su consistencia y no perjudicar la experiencia del usuario.

## El lado del servidor

A pesar de ser el actor principal en una aplicación web, el servidor es el lado pasivo de la comunicación, simplemente respondiendo a las solicitudes realizadas por el cliente. En la jerga web, *server* puede referirse a la máquina que recibe las solicitudes, el programa que maneja específicamente las solicitudes HTTP o la secuencia de comandos del destinatario que produce una respuesta a la solicitud. Esta última definición es la más relevante en el contexto de la arquitectura de aplicaciones web, pero todas están estrechamente relacionadas. Aunque solo están parcialmente dentro del alcance del desarrollador del servidor de aplicaciones, la máquina, el sistema operativo y el servidor HTTP no pueden ignorarse, porque son fundamentales para ejecutar el servidor de aplicaciones y, a menudo, se cruzan.

## Manejo de rutas de solicitudes

Los servidores HTTP, como Apache y NGINX, generalmente requieren cambios de configuración específicos para satisfacer las necesidades de la aplicación. De forma predeterminada, los servidores HTTP tradicionales asocian directamente la ruta indicada en la solicitud a un archivo en el sistema de archivos local. Si el servidor HTTP de un sitio web mantiene sus archivos HTML en el directorio `/srv/www`, por ejemplo, una solicitud con la ruta `/en/about.html` recibirá el contenido del archivo `/srv/www/en/about.html` como respuesta, si el archivo existe. Los sitios web más sofisticados, y especialmente las aplicaciones web, exigen tratamientos personalizados para diferentes tipos de solicitudes. En este escenario, parte de la implementación de la aplicación es modificar la configuración del servidor HTTP para cumplir con los requisitos de la aplicación.

Alternativamente, existen marcos que le permiten integrar la gestión de solicitudes HTTP y la implementación del código de la aplicación en un solo lugar, lo que permite al desarrollador centrarse más en el propósito de la aplicación que en los detalles de la plataforma. En Node.js Express, por ejemplo, todos los mapeos de solicitudes y la programación correspondiente se implementan mediante JavaScript. Como la programación de los clientes se realiza habitualmente en JavaScript, muchos desarrolladores consideran una buena idea desde la perspectiva del mantenimiento del código utilizar el mismo lenguaje para cliente y servidor. Otros lenguajes comúnmente utilizados para implementar el lado del servidor, ya sea en marcos o en servidores HTTP tradicionales, son PHP, Python, Ruby, Java y C #.

## Sistemas de gestión de bases de datos

Depende del criterio del equipo de desarrollo cómo se almacenan en el servidor los datos recibidos o solicitados por el cliente, pero existen pautas generales que se aplican a la mayoría de los casos. Es conveniente mantener el contenido estático (imágenes, código JavaScript y CSS que no cambian en el corto plazo) como archivos convencionales, ya sea en el propio sistema de archivos del servidor o distribuidos a través de una *red de entrega de contenido* (CDN). Otros tipos de contenido, como mensajes de correo electrónico en una aplicación de correo web, detalles de productos en una aplicación de compras y registros de transacciones, se almacenan de manera más conveniente en un *sistema de administración de base de datos* (DBMS).

El tipo más tradicional de sistema de gestión de bases de datos es la *base de datos relacional*. En él, el diseñador de la aplicación define las tablas de datos y el formato de entrada aceptado por cada tabla. El conjunto de tablas de la base de datos contiene todos los datos dinámicos consumidos y producidos por la aplicación. Una aplicación de compras, por ejemplo, puede tener una tabla que contiene una entrada con los detalles de cada producto en la tienda y una tabla que registra los artículos comprados por un usuario. La tabla de artículos comprados contiene referencias a entradas en la tabla de productos, creando relaciones entre las tablas. Este enfoque puede optimizar el almacenamiento y el acceso a los datos, así como permitir consultas en tablas combinadas utilizando

el lenguaje adoptado por el sistema de gestión de la base de datos. El lenguaje de base de datos relacional más popular es el *Structured Query Language* (SQL, pronunciado "sequel"), adoptado por las bases de datos de código abierto SQLite, MySQL, MariaDB y PostgreSQL.

Una alternativa a las bases de datos relacionales es una forma de base de datos que no requiere una estructura rígida para los datos. Estas bases de datos se denominan *bases de datos no relacionales* o simplemente *NoSQL*. Aunque pueden incorporar algunas características similares a las que se encuentran en las bases de datos relacionales, el enfoque está en permitir una mayor flexibilidad en el almacenamiento y acceso a los datos almacenados, pasando la tarea de procesar esos datos a la propia aplicación. MongoDB, CouchDB y Redis son sistemas comunes de administración de bases de datos no relacionales.

## Mantenimiento de contenido

Independientemente del modelo de base de datos adoptado, las aplicaciones deben agregar datos y probablemente actualizarlos durante la vida útil de las aplicaciones. En algunas aplicaciones, como el correo web, los propios usuarios proporcionan datos a la base de datos cuando utilizan el cliente para enviar y recibir mensajes. En otros casos, como en la aplicación de compras, es importante permitir que los mantenedores de la aplicación modifiquen la base de datos sin tener que recurrir a la programación. Por lo tanto, muchas organizaciones adoptan algún tipo de *sistema de gestión de contenido* (CMS), que permite a los usuarios no técnicos administrar la aplicación. Por lo tanto, para la mayoría de las aplicaciones web, es necesario implementar al menos dos tipos de clientes: un cliente no privilegiado, utilizado por usuarios normales, y un cliente privilegiado, utilizado por usuarios especiales para mantener y actualizar la información presentada por la aplicación.

# Ejercicios Guiados

1. ¿Qué lenguaje de programación se utiliza junto con HTML para crear clientes de aplicaciones web?

2. ¿En qué se diferencia la recuperación de una aplicación web de la de una aplicación nativa?

3. ¿En qué se diferencia una aplicación web de una aplicación nativa en el acceso al hardware local?

4. Cite una característica de un cliente de aplicación web que lo distinga de una página web normal.

# Ejercicios Exploratorios

1. ¿Qué característica ofrecen los navegadores modernos para superar el bajo rendimiento de los clientes de aplicaciones web con uso intensivo de CPU?

2. Si una aplicación web usa el paradigma REST para la comunicación cliente/servidor, ¿qué método HTTP debe usarse cuando el cliente solicita al servidor que borre un recurso específico?

3. Cite cinco lenguajes de secuencias de comandos de servidor compatibles con el servidor HTTP Apache.

4. ¿Por qué las bases de datos no relacionales se consideran más fáciles de mantener y actualizar que las bases de datos relacionales?

# Resumen

Esta lección cubre los conceptos y estándares en tecnología y arquitectura de desarrollo web. El principio es simple: el navegador web ejecuta la aplicación cliente, que se comunica con la aplicación principal que se ejecuta en el servidor. Aunque simples en principio, las aplicaciones web deben combinar muchas tecnologías para adoptar el principio de computación cliente y servidor en la web. La lección abarca los siguientes conceptos:

- El papel de los navegadores web y los servidores web.
- Tecnologías y estándares comunes de desarrollo web.
- Cómo los clientes web pueden comunicarse con el servidor.
- Tipos de servidores web y bases de datos de servidores.

# Respuestas a los ejercicios guiados

1. ¿Qué lenguaje de programación se utiliza junto con HTML para crear clientes de aplicaciones web?

   JavaScript

2. ¿En qué se diferencia la recuperación de una aplicación web de la de una aplicación nativa?

   No hay una aplicación web instalada. En cambio, algunas partes se ejecutan en el servidor y la interfaz del cliente se ejecuta en un navegador web normal.

3. ¿En qué se diferencia una aplicación web de una aplicación nativa en el acceso al hardware local?

   Todo acceso a los recursos locales, como almacenamiento, cámaras o micrófonos, está mediado por el navegador y requiere una autorización explícita del usuario para funcionar.

4. Cite una característica de un cliente de aplicación web que lo distinga de una página web normal.

   La interacción con las páginas web tradicionales se limita básicamente a hipervínculos y formularios de envío, mientras que los clientes de aplicaciones web están más cerca de una interfaz de aplicación convencional.

# Respuestas a los ejercicios de exploración

1. ¿Qué característica ofrecen los navegadores modernos para superar el bajo rendimiento de los clientes de aplicaciones web con uso intensivo de CPU?

   Los desarrolladores pueden utilizar WebAssembly para implementar las partes de la aplicación cliente que consumen mucha CPU. El código de WebAssembly generalmente tiene un mejor rendimiento que el JavaScript tradicional, porque requiere menos traducción de instrucciones.

2. Si una aplicación web usa el paradigma REST para la comunicación cliente/servidor, ¿qué método HTTP debe usarse cuando el cliente solicita al servidor que borre un recurso específico?

   REST se basa en métodos HTTP estándar, por lo que debería usar el método `DELETE` estándar en este caso.

3. Cite cinco lenguajes de secuencias de comandos de servidor compatibles con el servidor HTTP Apache.

   PHP, Go, Perl, Python y Ruby.

4. ¿Por qué las bases de datos no relacionales se consideran más fáciles de mantener y actualizar que las bases de datos relacionales?

   A diferencia de las bases de datos relacionales, las bases de datos no relacionales no requieren que los datos se ajusten a estructuras rígidas predefinidas, lo que facilita la implementación de cambios en las estructuras de datos sin afectar los datos existentes.




# 031.3 Conceptos básicos de HTTP

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 031.3

**Peso**

3

**Áreas de conocimiento clave**

- Comprender los métodos HTTP GET y POST, códigos de estado, encabezados y tipos de contenido
- Comprender la diferencia entre contenido estático y dinámico.
- Comprender las URL de HTTP
- Comprender cómo se asignan las URL HTTP a las rutas del sistema de archivos
- Subir archivos a la raíz de documentos de un servidor web
- Comprender el almacenamiento en caché
- Comprender las cookies
- Conciencia de sesiones y secuestro de sesiones.
- Conocimiento de los servidores HTTP de uso común
- Conocimiento de HTTPS y TLS
- Conocimiento de los sockets web
- Conocimiento de hosts virtuales
- Conocimiento de los servidores HTTP comunes
- Conocimiento de los requisitos y limitaciones de ancho de banda y latencia de la red

### Lista parcial de archivos, términos y utilidades

- GET, POST
- 200, 301, 302, 401, 403, 404, 500
- Apache HTTP Server (httpd), NGINX




# 031.3 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 031 Desarrollo de software y tecnologías web |
| **Objetivo:** | 031.3 Conceptos básicos de HTTP |
| **Lección:** | 1 de 1 |

## Introducción

El *HyperText Transfer Protocol* (HTTP) define cómo un cliente solicita al servidor un recurso específico. Su principio de funcionamiento es bastante simple: el cliente crea un mensaje de solicitud identificando el recurso que necesita y reenvía ese mensaje al servidor a través de la red. A su vez, el servidor HTTP evalúa dónde extraer el recurso solicitado y envía un mensaje de respuesta al cliente. El mensaje de respuesta contiene detalles sobre el recurso solicitado, seguido del recurso en sí.

Más específicamente, HTTP es el conjunto de reglas que definen cómo la aplicación cliente debe formatear los mensajes *request* que se enviarán al servidor. Luego, el servidor sigue las reglas HTTP para interpretar la solicitud y formatear los mensajes *reply*. Además de solicitar o transferir el contenido solicitado, los mensajes HTTP contienen información adicional sobre el cliente y el servidor involucrados, sobre el contenido en sí e incluso sobre su indisponibilidad. Si no se puede enviar un recurso, un código en la respuesta explica el motivo de la indisponibilidad y, si es posible, indica hacia dónde se movió el recurso.

La parte del mensaje que define los detalles del recurso y otra información de contexto se denomina *header*. La parte que sigue al encabezado, y que contiene el contenido del recurso correspondiente, se llama *payload* del mensaje. Tanto los mensajes de solicitud como los de respuesta pueden tener un

*payload*, pero en la mayoría de los casos, solo el mensaje de respuesta tiene uno.

## Solicitud del cliente

La primera etapa de un intercambio de datos HTTP entre el cliente y el servidor es iniciada por el primero, cuando escribe un mensaje de solicitud en el servidor. Tomemos, por ejemplo, una tarea común del navegador: cargar una página HTML desde un servidor que aloja un sitio web, como `https://learning.lpi.org/en/`. La dirección, o URL, proporciona varios datos relevantes. En este ejemplo en particular, aparecen tres datos:

- El protocolo: Protocolo de transferencia de hipertexto seguro (`https`), una versión encriptada de HTTP.
- El nombre de dominio del proveedor de alojamiento web (`learning.lpi.org`)
- La ubicación del recurso solicitado en el servidor (el directorio `/en/`—en este caso, la versión en inglés de la página de inicio).

**NOTE** Un *Uniform Resource Locator* (URL) es una dirección que apunta a un recurso en Internet. Este recurso suele ser un archivo que se puede copiar desde un servidor remoto, pero las URL también pueden indicar contenido generado dinámicamente y flujos de datos.

### Cómo maneja el cliente la URL

Antes de contactar al servidor, el cliente necesita convertir `learning.lpi.org` a su dirección IP correspondiente. El cliente utiliza otro servicio de Internet, el *Domain Name System* (DNS), para solicitar la dirección IP de un nombre de host de uno o más servidores DNS predefinidos (los servidores DNS suelen definirse automáticamente por el proveedor de servicios de Internet, ISP).

Con la dirección IP del servidor, el cliente intenta conectarse al puerto HTTP o HTTPS. Los puertos de red son números de identificación definidos por el *Transmission Control Protocol* (TCP) para entrelazar e identificar distintos canales de comunicación dentro de una conexión cliente/servidor. De forma predeterminada, los servidores HTTP reciben solicitudes en los puertos TCP 80 (HTTP) y 443 (HTTPS).

**NOTE** Existen otros protocolos utilizados por las aplicaciones web para implementar la comunicación cliente/servidor. Para llamadas de audio y video, por ejemplo, es más apropiado usar WebSockets, un protocolo de nivel inferior que es más eficiente que HTTP para transferir flujos de datos en ambas direcciones.

El formato del mensaje de solicitud que el cliente envía al servidor es el mismo en HTTP y HTTPS.

HTTPS ya se usa más ampliamente que HTTP, porque todos los intercambios de datos entre el cliente y el servidor están encriptados, lo cual es una característica indispensable para promover la privacidad y la seguridad en las redes públicas. La conexión cifrada se establece entre el cliente y el servidor incluso antes de que se intercambie cualquier mensaje HTTP, utilizando el protocolo criptográfico *Transport Layer Security* (TLS). Al hacer esto, TLS encapsula toda la comunicación HTTPS. Una vez descifrada, la solicitud o respuesta transmitida a través de HTTPS no es diferente de una solicitud o respuesta realizada exclusivamente a través de HTTP.

El tercer elemento de nuestra URL, `/en/`, será interpretado por el servidor como la ubicación o ruta del recurso que se solicita. Si la ruta no se proporciona en la URL, se utilizará la ubicación predeterminada `/`. La implementación más simple de un servidor HTTP asocia rutas en URL con archivos en el sistema de archivos donde se ejecuta el servidor, pero esta es solo una de las muchas opciones disponibles en servidores HTTP más sofisticados.

### El mensaje de solicitud

HTTP opera a través de una conexión ya establecida entre cliente y servidor, generalmente implementada en TCP y cifrada con TLS. De hecho, una vez que una conexión que cumple con los requisitos impuestos por el servidor está lista, una solicitud HTTP escrita a mano en texto sin formato podría generar la respuesta del servidor. En la práctica, sin embargo, los programadores rara vez necesitan implementar rutinas para componer mensajes HTTP, ya que la mayoría de los lenguajes de programación proporcionan mecanismos que automatizan éste proceso. En el caso de la URL de ejemplo, `https://learning.lpi.org/en/`, el mensaje de solicitud más simple posible tendría el siguiente contenido:

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: text/html
```

La primera palabra de la primera línea identifica el *método* HTTP. Define qué operación desea realizar el cliente en el servidor. El método GET informa al servidor que el cliente solicita el recurso que le sigue: `/en/`. Tanto el cliente como el servidor pueden admitir más de una versión del protocolo HTTP, por lo que la versión que se adoptará en el intercambio de datos también se proporciona en la primera línea: `HTTP/1.1`.

> **NOTE** La versión más reciente del protocolo HTTP es HTTP/2. Entre otras diferencias, los mensajes escritos en HTTP/2 se codifican en una estructura binaria, mientras que los mensajes escritos en HTTP/1.1 se envían en texto sin formato. Este cambio optimiza las velocidades de transmisión de datos, pero el contenido de los mensajes es básicamente el mismo.

El encabezado puede contener más líneas después de la primera para contextualizar y ayudar a identificar la solicitud al servidor. El campo de encabezado `Host`, por ejemplo, puede parecer redundante, porque el host del servidor obviamente ha sido identificado por el cliente para establecer la conexión y es razonable suponer que el servidor conoce su propia identidad. No obstante, es importante informar al host del nombre de host esperado en el encabezado de la solicitud, porque es una práctica común utilizar el mismo servidor HTTP para alojar más de un sitio web. (En este escenario, cada host específico se denomina *virtual host*). Por lo tanto, el servidor HTTP utiliza el campo `Host` para identificar a cuál se refiere la solicitud.

El campo de encabezado `User-Agent` contiene detalles sobre el programa cliente que realiza la solicitud. Este campo puede ser utilizado por el servidor para adaptar la respuesta a las necesidades de un cliente específico, pero se utiliza más a menudo para producir estadísticas sobre los clientes que utilizan el servidor.

El campo `Accept` tiene un valor más inmediato, porque informa al servidor sobre el formato del recurso solicitado. Si el cliente es indiferente sobre el formato del recurso, el campo `Accept` puede especificar `*/*` como el formato predefinido.

Hay muchos otros campos de encabezado que se pueden usar en un mensaje HTTP, pero los campos que se muestran en el ejemplo son suficientes para solicitar un recurso del servidor.

Además de los campos del encabezado de la solicitud, el cliente puede incluir otros datos complementarios en la solicitud HTTP que se enviará al servidor. Si estos datos constan solo de parámetros de texto simples, en el formato `name=value`, se pueden agregar a la ruta del método GET. Los parámetros están incrustados en la ruta después de un signo de interrogación y están separados por el caracter (&):

```
GET /cgi-bin/receive.cgi?name=LPI&email=info@lpi.org HTTP/1.1
```

En este ejemplo, `/cgi-bin/receive.cgi` es la ruta al script en el servidor que procesará y posiblemente usará los parámetros `name` e `email`, obtenidos de la ruta de la solicitud. La cadena que corresponde a los campos, en el formato `name=LPI&email=info@lpi.org`, se llama *query string* y es suministrada al script `receive.cgi` por el servidor HTTP que recibe la solicitud.

Cuando los datos se componen de más que campos de texto cortos, es más apropiado enviarlos en el payload del mensaje. En este caso, se debe utilizar el método HTTP POST para que el servidor reciba y procese el payload del mensaje, de acuerdo con las especificaciones indicadas en el encabezado de la solicitud. Cuando se utiliza el método POST, el encabezado de la solicitud debe proporcionar el tamaño del payload que se enviará a continuación y cómo se formatea el cuerpo:

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
Content-Length: 1503
Content-Type: multipart/form-data; boundary=------------------------
405f7edfd646a37d
```

El campo `Content-Length` indica el tamaño en bytes de la carga útil y el campo `Content-Type` indica su formato. El formato `multipart/form-data` es el más comúnmente utilizado en los formularios HTML tradicionales que utilizan el método POST. En este formato, cada campo insertado en el payload de la solicitud está separado por el código indicado por la palabra clave `boundary`. El método POST debe usarse solo cuando sea apropiado, ya que usa una cantidad de datos ligeramente mayor que una solicitud equivalente realizada con el método GET. Debido a que el método GET envía los parámetros directamente en el encabezado del mensaje de la solicitud, el intercambio total de datos tiene una latencia más baja, ya que no será necesaria una etapa de conexión adicional para transmitir el cuerpo del mensaje.

## El encabezado de respuesta

Una vez que el servidor HTTP recibe el encabezado del mensaje de solicitud, el servidor devuelve un mensaje de respuesta al cliente. Una solicitud de archivo HTML normalmente tiene un encabezado de respuesta como este:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 18170
Content-Type: text/html
Date: Mon, 05 Apr 2021 13:44:25 GMT
Etag: "606adcd4-46fa"
Last-Modified: Mon, 05 Apr 2021 09:48:04 GMT
Server: nginx/1.17.10
```

La primera línea proporciona la versión del protocolo HTTP utilizada en el mensaje de respuesta, que debe corresponder a la versión utilizada en el encabezado de la solicitud. Luego, todavía en la primera línea, aparece el código de estado de la respuesta, indicando cómo el servidor interpretó y

generó la respuesta a la solicitud.

El código de estado es un número de tres dígitos, donde el dígito más a la izquierda define la clase de respuesta. Hay cinco clases de códigos de estado, numerados del 1 al 5, cada uno de los cuales indica un tipo de acción realizada por el servidor:

**1xx (Informational)**
: Se recibió la solicitud, continuando el proceso.

**2xx (Successful)**
: La solicitud fue recibida, comprendida y aceptada con éxito.

**3xx (Redirection)**
: Es necesario realizar más acciones para completar la solicitud.

**4xx (Client Error)**
: La solicitud contiene una sintaxis incorrecta o no se puede cumplir.

**5xx (Server Error)**
: El servidor no cumplió con una solicitud aparentemente válida.

El segundo y tercer dígitos se utilizan para indicar detalles adicionales. El código 200, por ejemplo, indica que la solicitud podría ser respondida sin problemas. Como se muestra en el ejemplo, también se puede proporcionar una breve descripción de texto después del código de respuesta (`OK`). Algunos códigos específicos son de particular interés para garantizar que el cliente HTTP pueda acceder al recurso en situaciones adversas o para ayudar a identificar el motivo del error en caso de que una solicitud no sea satisfactoria:

**`301 Moved Permanently`**
: Al recurso de destino se le ha asignado una nueva URL permanente, proporcionada por el campo de encabezado `Location` en la respuesta.

**`302 Found`**
: El recurso de destino reside temporalmente en una URL diferente.

**`401 Unauthorized`**
: La solicitud no se ha aplicado porque carece de credenciales de autenticación válidas para el recurso de destino.

**`403 Forbidden`**

La respuesta `Forbidden` indica que, aunque la solicitud es válida, el servidor está configurado para no proporcionarla.

**`404 Not Found`**

El servidor de origen no encontró una representación actual para el recurso de destino o no está dispuesto a revelar que existe.

**`500 Internal Server Error`**

El servidor encontró una condición inesperada que le impidió cumplir con la solicitud.

**`502 Bad Gateway`**

El servidor, mientras actuaba como puerta de enlace o proxy, recibió una respuesta no válida de un servidor entrante al que accedió mientras intentaba cumplir con la solicitud.

Aunque indican que no fue posible cumplir con la solicitud, los códigos de estado `4xx` y `5xx` al menos indican que el servidor HTTP se está ejecutando y es capaz de recibir solicitudes. Los códigos `4xx` requieren que se realice una acción en el cliente, porque su URL o credenciales son incorrectas. Por el contrario, los códigos `5xx` indican algo incorrecto en el lado del servidor. Por lo tanto, en el contexto de las aplicaciones web, estas dos clases de códigos de estado indican que el origen del error se encuentra en la propia aplicación, ya sea cliente o servidor, no en la infraestructura subyacente.

## Contenido estático y dinámico

Los servidores HTTP utilizan dos mecanismos básicos para cumplir con el contenido solicitado por el cliente. El primer mecanismo proporciona *contenido estático*: es decir, la ruta indicada en el mensaje de solicitud corresponde a un archivo en el sistema de archivos local del servidor. El segundo mecanismo proporciona *contenido dinámico*: es decir, el servidor HTTP reenvía la solicitud a otro programa, probablemente un script, para generar la respuesta a partir de diferentes fuentes, como bases de datos y otros archivos.

Aunque existen diferentes servidores HTTP, todos utilizan el mismo protocolo de comunicación HTTP y adoptan más o menos las mismas convenciones. Una aplicación que no tiene una necesidad específica se puede implementar con cualquier servidor tradicional, como Apache o NGINX. Ambos son capaces de generar contenido dinámico y proporcionar contenido estático, pero existen sutiles diferencias en la configuración de cada uno.

La ubicación de los archivos estáticos que se entregarán, por ejemplo, se define de diferentes formas en Apache y NGINX. La convención es mantener estos archivos en un directorio específico para este propósito, con un nombre asociado con el host, por ejemplo, `/var/www/learning.lpi.org/`. En Apache, esta ruta está definida por la directiva de configuración `DocumentRoot`

`/var/www/learning.lpi.org`, en una sección que define un host virtual. En NGINX, la directiva utilizada es `root /var/www/learning.lpi.org` en la sección de `server` del archivo de configuración.

Cualquiera que sea el servidor que elija, los archivos en `/var/www/learning.lpi.org/` se servirán a través de HTTP casi de la misma manera. Algunos campos en el encabezado de respuesta y su contenido pueden variar entre los dos servidores, pero campos como `Content-Type` deben estar presentes en el encabezado de respuesta y deben ser consistentes en cualquier servidor.

## Almacenamiento en caché

HTTP fue diseñado para funcionar en cualquier tipo de conexión a Internet, rápida o lenta. Además, la mayoría de los intercambios HTTP tienen que atravesar muchos nodos de red debido a la arquitectura distribuida de Internet. Como resultado, es importante adoptar alguna estrategia de almacenamiento en caché para evitar la transferencia redundante de contenido descargado previamente. Las transferencias HTTP pueden funcionar con dos tipos básicos de caché: *shared* y *private*.

Más de un cliente utiliza una caché compartida. Por ejemplo, un gran proveedor de contenido puede utilizar cachés en servidores distribuidos geográficamente, de modo que los clientes obtengan los datos de su servidor más cercano. Una vez que un cliente ha realizado una solicitud y su respuesta se almacenó en una caché compartida, otros clientes que realicen la misma solicitud en esa misma área recibirán la respuesta en caché.

El propio cliente crea una caché privada para su uso exclusivo. Es el tipo de almacenamiento en caché que hace el navegador web para imágenes, archivos CSS, JavaScript o el documento HTML en sí, por lo que no es necesario volver a descargarlos si se solicita en un futuro próximo.

**NOTE** No todas las solicitudes HTTP deben almacenarse en caché. Una solicitud que utiliza el método POST, por ejemplo, implica una respuesta asociada exclusivamente con esa solicitud en particular, por lo que su contenido de respuesta no debe reutilizarse. De forma predeterminada, solo se almacenan en caché las respuestas a las solicitudes realizadas mediante el método GET. Además, solo las respuestas con códigos de estado concluyentes como 200 (OK), 206 (Partial Content), 301 (Moved Permanently) y 404 (Not Found) son adecuadas para el almacenamiento en caché.

Tanto la estrategia de caché compartida como la privada utilizan encabezados HTTP para controlar cómo se debe almacenar el contenido descargado. Para la caché privada, el cliente consulta el encabezado de respuesta y verifica si el contenido de la caché local todavía corresponde al contenido remoto actual. Si lo hace, el cliente renuncia a la transferencia del payload de respuesta y usa la versión local.

La validez del recurso almacenado en caché se puede evaluar de varias formas. El servidor puede proporcionar una fecha de vencimiento en el encabezado de respuesta para la primera solicitud, de modo que el cliente descarta el recurso almacenado en caché al final del plazo y lo solicita nuevamente para obtener la versión actualizada. Sin embargo, el servidor no siempre puede determinar la fecha de vencimiento de un recurso, por lo que es común usar el campo de encabezado de respuesta `ETag` para identificar la versión del recurso, por ejemplo, `Etag: "606adcd4-46fa"`.

Para verificar que un recurso almacenado en caché necesita actualizarse, el cliente solicita solo su encabezado de respuesta del servidor. Si el campo `ETag` coincide con el de la versión almacenada localmente, el cliente reutiliza el contenido almacenado en caché. De lo contrario, el contenido actualizado del recurso se descarga del servidor.

## Sesiones HTTP

En un sitio web o aplicación web convencional, las funciones que manejan el control de sesión se basan en encabezados HTTP. El servidor no puede asumir, por ejemplo, que todas las solicitudes que provienen de la misma dirección IP son del mismo cliente. El método más tradicional que permite al servidor asociar diferentes solicitudes a un solo cliente es el uso de *cookies*, una etiqueta de identificación que le da el servidor al cliente y que se proporciona en el encabezado HTTP.

Las cookies permiten al servidor conservar información sobre un cliente específico, incluso si la persona que ejecuta el cliente no se identifica explícitamente. Con las cookies, es posible implementar sesiones donde los inicios de sesión, tarjetas de compra, preferencias, etc., se conservan entre diferentes solicitudes realizadas al mismo servidor que las proporcionó. Las cookies también se utilizan para rastrear la navegación del usuario, por lo que es importante solicitar su consentimiento antes de enviarlas.

El servidor establece la cookie en el encabezado de respuesta utilizando el campo `Set-Cookie`. El valor del campo es un par `name=value` elegido para representar algún atributo asociado con un cliente específico. El servidor puede, por ejemplo, crear un número de identificación para un cliente que solicita un recurso por primera vez y pasarlo al cliente en el encabezado de respuesta:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Set-Cookie: client_id=62b5b719-fcbf
```

Si el cliente permite el uso de cookies, las nuevas solicitudes a este mismo servidor tienen el campo de cookies en el encabezado:

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
Cookie: client_id=62b5b719-fcbf
```

Con este número de identificación, el servidor puede recuperar definiciones específicas para el cliente y generar una respuesta personalizada. También es posible utilizar más de un campo `Set-Cookie` para entregar diferentes cookies al mismo cliente. De esta forma, se puede conservar más de una definición en el lado del cliente.

Las cookies plantean tanto problemas de privacidad como posibles agujeros de seguridad, porque existe la posibilidad de que se puedan transferir a otro cliente, que será identificado por el servidor como el cliente original. Las cookies utilizadas para conservar sesiones pueden dar acceso a información confidencial del cliente original. Por tanto, es muy importante que los clientes adopten mecanismos de protección local para evitar que sus cookies sean extraídas y reutilizadas sin autorización.

## Ejercicios guiados

1. ¿Qué método HTTP utiliza el siguiente mensaje de solicitud?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

2. Cuando un servidor HTTP aloja muchos sitios web, ¿cómo puede identificar para cuál es una solicitud?

3. ¿Qué parámetro proporciona la cadena de consulta de la URL `https://www.google.com/search?q=LPI`?

4. ¿Por qué la siguiente solicitud HTTP no es adecuada para el almacenamiento en caché?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

## Ejercicios de exploración

1. ¿Cómo podría usar el navegador web para monitorear las solicitudes y respuestas hechas por una página HTML?

2. Los servidores HTTP que proporcionan contenido estático generalmente asignan la ruta solicitada a un archivo en el sistema de archivos del servidor. ¿Qué sucede cuando la ruta de la solicitud apunta a un directorio?

3. El contenido de los archivos enviados a través de HTTPS está protegido por cifrado, por lo que las computadoras entre el cliente y el servidor no pueden leerlos. A pesar de esto, ¿pueden estas computadoras intermedias identificar qué recurso ha solicitado el cliente al servidor?

## Resumen

Esta lección cubre los conceptos básicos de HTTP, el protocolo principal utilizado por las aplicaciones cliente para solicitar recursos de los servidores web. La lección abarca los siguientes conceptos:

- Solicitar mensajes, campos de encabezado y métodos.
- Códigos de estado de respuesta.
- Cómo los servidores HTTP generan respuestas.
- Funciones HTTP útiles para el almacenamiento en caché y la gestión de sesiones.

## Respuestas a los ejercicios guiados

1. ¿Qué método HTTP utiliza el siguiente mensaje de solicitud?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

   El método POST.

2. Cuando un servidor HTTP aloja muchos sitios web, ¿cómo puede identificar para cuál es una solicitud?

   El campo `Host` en el encabezado de la solicitud proporciona el sitio web de destino.

3. ¿Qué parámetro proporciona la cadena de consulta de la URL `https://www.google.com/search?q=LPI`?

   El parámetro denominado `q` con el valor `LPI`.

4. ¿Por qué la siguiente solicitud HTTP no es adecuada para el almacenamiento en caché?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

   Debido a que las solicitudes realizadas con el método POST implican una operación de escritura en el servidor, no deben almacenarse en caché.

## Respuestas a los ejercicios de exploración

1. ¿Cómo podría usar el navegador web para monitorear las solicitudes y respuestas hechas por una página HTML?

   Todos los navegadores populares ofrecen *herramientas de desarrollo* que, entre otras cosas, pueden mostrar todas las transacciones de red que se han realizado en la página actual.

2. Los servidores HTTP que proporcionan contenido estático generalmente asignan la ruta solicitada a un archivo en el sistema de archivos del servidor. ¿Qué sucede cuando la ruta de la solicitud apunta a un directorio?

   Depende de cómo esté configurado el servidor. De forma predeterminada, la mayoría de los servidores HTTP buscan un archivo llamado `index.html` (u otro nombre predefinido) en ese mismo directorio y lo envían como respuesta. Si el archivo no está allí, el servidor emite una respuesta `404 Not Found`.

3. El contenido de los archivos enviados a través de HTTPS está protegido por cifrado, por lo que las computadoras entre el cliente y el servidor no pueden leerlos. A pesar de esto, ¿pueden estas computadoras intermedias identificar qué recurso ha solicitado el cliente al servidor?

   No, porque los encabezados HTTP de solicitud y respuesta también están encriptados por TLS.




# Tema 032: Marcado de documentos HTML




# 032.1 Anatomía del documento HTML

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 032.1

**Peso**

2

**Áreas de conocimiento clave**

- Crear un documento HTML simple
- Comprender el papel de HTML
- Comprender el esqueleto HTML
- Comprender la sintaxis HTML (etiquetas, atributos, comentarios)
- Comprender el encabezado HTML
- Comprender las metaetiquetas
- Comprender la codificación de caracteres

**Lista parcial de archivos, términos y utilidades**

- `<!DOCTYPE html>`
- `<html>`
- `<head>`
- `<body>`
- `<meta>`, incluido el juego de caracteres (UTF-8), el nombre y los atributos de contenido




# 032.1 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 032 Marcado de documentos HTML |
| **Objetivo:** | 032.1 Anatomía del documento HTML |
| **Lección:** | 1 de 1 |

## Introducción

HTML (*HyperText Markup Language*) es un lenguaje de marcado que le especifica a los navegadores web cómo estructurar y mostrar páginas web. La versión actual es la 5.0, que se lanzó en 2012. La sintaxis HTML está definida por el *World Wide Web Consortium* (W3C).

HTML es una habilidad fundamental en el desarrollo web, ya que define la estructura y gran parte de la apariencia de un sitio web. Si desea una carrera en desarrollo web, HTML es definitivamente un buen punto de partida.

### Anatomía de un documento HTML

Una página HTML básica tiene la siguiente estructura:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My HTML Page</title>
    <!-- This is the Document Header -->
  </head>

  <body>
    <!-- This is the Document Body -->
  </body>
</html>
```

Ahora, analicemos en detalle.

### Etiquetas HTML

HTML usa *elementos* y *etiquetas* para describir y dar formato al contenido. Las etiquetas constan de paréntesis angulares alrededor de un nombre de etiqueta, por ejemplo, `<title>`. El nombre de la etiqueta no distingue entre mayúsculas y minúsculas, aunque el World Wide Web Consortium (W3C) recomienda usar letras minúsculas en las versiones actuales de HTML. Estas etiquetas se utilizan para crear elementos HTML. `<title>` es un ejemplo de una *etiqueta de apertura* de un elemento HTML que define el título de un documento HTML. Sin embargo, un elemento tiene dos componentes más. Un elemento `<title>` completo con este aspecto:

```
<title>My HTML Page</title>
```

Aquí, `My HTML Page` sirve como elemento *contenido*, mientras que `</title>` sirve como *etiqueta de cierre* que declara que este elemento está completo.

**NOTE** No es necesario cerrar todos los elementos HTML; en tales casos, hablamos de elementos vacíos y elementos de cierre automáticos.

Estos son los otros elementos HTML del ejemplo anterior:

**`<html>`**

Incluye todo el documento HTML. Contiene todas las etiquetas que componen la página. También indica que el contenido de este archivo está en lenguaje HTML. Su etiqueta de cierre correspondiente es `</html>`.

**`<head>`**

Un contenedor para toda la metainformación relacionada con la página. La etiqueta de cierre correspondiente de este elemento es `</head>`

**`<body>`**

Un contenedor para el cuerpo de la página y su representación estructural. Su etiqueta de cierre correspondiente es `</body>`.

Las etiquetas `<html>`, `<head>`, `<body>` y `<title>` son las denominadas *etiquetas esqueleto*, que proporcionan la estructura básica de un documento HTML. En particular, le especifican al navegador web que está leyendo una página HTML.

**NOTE** De estos elementos HTML, el único que se requiere para validar un documento HTML es la etiqueta `<title>`.

Como puede ver, cada página HTML es un documento bien estructurado e incluso podría denominarse un árbol, donde el elemento `<html>` representa la raíz del documento y los elementos `<head>` y `<body>` son las primeras ramas. El ejemplo muestra que es posible anidar elementos: por ejemplo, el elemento `<title>` está anidado dentro del elemento `<head>`, que a su vez está anidado dentro del elemento `<html>`.

Para asegurarse de que su código HTML sea legible y mantenible, asegúrese de que todos los elementos HTML estén cerrados correctamente y en orden. Los navegadores web aún pueden mostrar su sitio web como se esperaba, pero el anidamiento incorrecto de elementos y sus etiquetas es una práctica propensa a errores.

Finalmente, una mención especial va a la declaración *doctype* en la parte superior de la estructura del documento de ejemplo. `<! DOCTYPE>` no es una etiqueta HTML, sino una instrucción para el navegador web que especifica la versión HTML utilizada en el documento. En la estructura básica del documento HTML mostrada anteriormente, se usó `<! DOCTYPE html>`, especificando que se usa HTML5 en este documento.

## Comentarios HTML

Al crear una página HTML, es una buena práctica insertar comentarios en el código para mejorar su legibilidad y describir el propósito de bloques de código más grandes. Se inserta un comentario entre las etiquetas `<!--` y `-->`, como se muestra en el siguiente ejemplo:

```
<!-- This is a comment. -->

<!--
    This is a
    multiline
    comment.
-->
```

El ejemplo demuestra que los comentarios HTML se pueden colocar en una sola línea, pero también pueden abarcar varias. En cualquier caso, el resultado es que el texto entre `<!--` y `-->` es ignorado por el navegador web y, por lo tanto, no se muestra en la página HTML. Con base en estas consideraciones, puede deducir que la página HTML básica que se muestra en el párrafo “Anatomía de un documento HTML” no muestra ningún texto, porque las líneas `<!-- This is the Document Header -->` y `<!-- This is the Document Body -->` son solo dos comentarios.

**WARNING** Los comentarios no se pueden anidar.

### Atributos HTML

Las etiquetas HTML pueden incluir uno o más *atributos* para especificar detalles del elemento HTML. Una etiqueta simple con dos atributos tiene la siguiente forma:

```
<tag attribute-a="value-a" attribute-b="value-b">
```

Los atributos siempre deben establecerse en la etiqueta de apertura.

Un atributo consta de un nombre, que indica la propiedad que se debe establecer, un signo igual y el valor deseado entre comillas. Tanto las comillas simples como las dobles son aceptables, pero se recomienda utilizar comillas simples o dobles de forma coherente durante todo el proyecto. Es importante no mezclar comillas simples y dobles para un solo valor de atributo, ya que el navegador web no reconocerá las comillas mixtas como una sola unidad.

**NOTE** Puede incluir un tipo de comillas dentro del otro tipo sin ningún problema. Por ejemplo, si necesita usar `'` en un valor de atributo, puede envolver ese valor dentro de `"`. Sin embargo, si desea usar el mismo tipo de comillas dentro del valor que está usando para envolver el valor, necesita usar `"` para `"` y `'` para `'`.

Los atributos se pueden categorizar en *atributos centrales* y *atributos específicos* como se explica en las siguientes secciones.

## Atributos centrales

Los atributos centrales son atributos que se pueden utilizar en cualquier elemento HTML. Incluyen:

`title`
: Describe el contenido del elemento. Su valor a menudo se muestra como una información sobre herramientas que se muestra cuando el usuario mueve el cursor sobre el elemento.

`id`
: Asocia un identificador único con un elemento. Este identificador debe ser único dentro del documento, y el documento no se validará cuando varios elementos compartan el mismo `id`.

`style`
: Asigna propiedades gráficas (estilos CSS) al elemento.

`class`
: Especifica una o varias clases para el elemento en una lista de nombres de clases separados por espacios. Se puede hacer referencia a estas clases en hojas de estilo CSS.

`lang`
: Especifica el idioma del contenido del elemento mediante códigos de idioma de dos caracteres estándar ISO-639.

**NOTE** El desarrollador puede almacenar información personalizada sobre un elemento definiendo un atributo llamado `data-`, que se indica anteponiendo el nombre deseado con `data-` como en `data-additionalinfo`. Puede asignar a este atributo un valor como cualquier otro atributo.

## Atributos específicos

Otros atributos son específicos de cada elemento HTML. Por ejemplo, el atributo `src` de un elemento HTML `<img>` especifica la URL de una imagen. Hay muchos atributos más específicos, que se tratarán en las siguientes lecciones.

## Encabezado del documento

El encabezado del documento define la metainformación con respecto a la página y se describe mediante el elemento `<head>`. De forma predeterminada, el navegador web no muestra la información del encabezado del documento. Si bien es posible utilizar el elemento `<head>` para contener elementos HTML que podrían mostrarse en la página, no se recomienda hacerlo.

### Título

El título del documento se especifica mediante el elemento `<title>`. El título definido entre las etiquetas aparece en la barra de título del navegador (web bowser) y es el nombre sugerido para los marcadores (bookmarks) cuando agrega a favoritos la página. También se muestra en los resultados del motor de búsqueda como el título de la página.

Un ejemplo de este elemento es el siguiente:

```
<title>My test page</title>
```

La etiqueta `<title>` es obligatoria en todos los documentos HTML y debe aparecer solo una vez en cada documento.

> **NOTE** No confunda el título del documento con el encabezado de la página, que se encuentra en el cuerpo.

### Metadatos

El elemento `<meta>` se utiliza para especificar la metainformación para describir con más detalle el contenido de un documento HTML. Es un elemento de cierre automático, lo que significa que no tiene etiqueta de cierre. Aparte de los atributos básicos que son válidos para cada elemento HTML, el elemento `<meta>` también usa los siguientes atributos:

**`name`**
: Define qué metadatos se describirán en este elemento. Se puede establecer en cualquier valor personalizado definido, pero los valores más utilizados son `autor`, `descripción` y `palabras clave`.

**`http-equiv`**
: Proporciona un encabezado HTTP para el valor del atributo `content`. Un valor común es `refresh`, que se explicará más adelante. Si se establece este atributo, no se debe establecer el atributo `name`.

**`content`**
: Proporciona el valor asociado con el atributo `name` o `http-equiv`.

**`charset`**
: Especifica la codificación de caracteres para el documento HTML, por ejemplo, `utf-8` para configurarlo en formato de transformación Unicode: 8 bits.

## Agregar un autor, una descripción y palabras clave

Usando la etiqueta `<meta>`, puede especificar información adicional sobre el autor de la página HTML y describir el contenido de la página de esta manera:

```
<meta name="author" content="Name Surname">
<meta name="description" content="A short summary of the page content.">
```

Intente incluir una serie de palabras claves relacionadas con el contenido de la página en la descripción. Esta descripción es a menudo lo primero que ve un usuario cuando navega con un motor de búsqueda.

Si también desea proporcionar palabras claves adicionales relacionadas con la página web a los motores de búsqueda, puede agregar este elemento:

```
<meta name="keywords" content="keyword1, keyword2, keyword3, keyword4, keyword5">
```

**NOTE** En el pasado, los spammers ingresaban cientos de palabras clave y descripciones no relacionadas con el contenido real de la página, de modo que también aparecían en búsquedas no relacionadas con los términos que buscaban las personas. Hoy en día, las etiquetas `<meta>` quedan relegadas a un lugar de importancia secundaria y se utilizan únicamente para consolidar los temas tratados en la página web, por lo que ya no es posible engañar a los nuevos y más sofisticados algoritmos de los buscadores.

## Redirigir una página HTML y definir un intervalo de tiempo para que el documento se actualice.

Con la etiqueta `<meta>`, puede actualizar automáticamente una página HTML después de un período determinado (por ejemplo, después de 30 segundos) de esta manera:

```
<meta http-equiv="refresh" content="30">
```

Alternativamente, puede redirigir una página web a otra página web después de la misma cantidad de tiempo con el siguiente código:

```
<meta http-equiv="refresh" content="30; url=http://www.lpi.org">
```

En este ejemplo, el usuario es redirigido desde la página actual a `http://www.lpi.org` después de 30 segundos. Los valores pueden ser los que desee. Por ejemplo, si especifica `content="0; url=http://www.lpi.org"`, la página se redirige inmediatamente.

## Especificar la codificación de caracteres

El atributo `charset` especifica la codificación de caracteres para el documento HTML. Un ejemplo común es:

```
<meta charset="utf-8">
```

Este elemento especifica que la codificación de caracteres del documento es `utf-8`, que es un conjunto de caracteres universal que incluye prácticamente cualquier caracter de cualquier lenguaje humano. Por lo tanto, al usarlo, evitará problemas al mostrar algunos caracteres que pueda tener al usar otros conjuntos de caracteres como ISO-8859-1 (el alfabeto latino).

## Otros ejemplos útiles

Otras dos aplicaciones útiles de la etiqueta `<meta>` son:

- Configurar cookies para realizar un seguimiento de los visitantes del sitio.
- Tome el control de la ventana gráfica (el área visible de una página web dentro de la ventana de un navegador web), que depende del tamaño de la pantalla del dispositivo del usuario (por ejemplo, un teléfono móvil o una computadora).

Sin embargo, estos dos ejemplos están más allá del alcance del examen y su estudio se deja al lector curioso para que lo explore en otro lugar.

## Ejercicios guiados

1. Para cada una de las siguientes etiquetas, indique la etiqueta de cierre correspondiente:

| | |
|---|---|
| `<body>` | |
| `<head>` | |
| `<html>` | |
| `<meta>` | |
| `<title>` | |

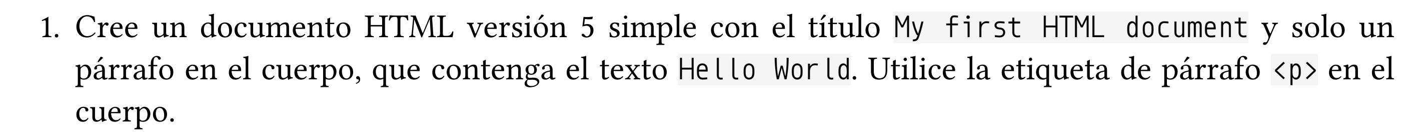
2. ¿Cuál es la diferencia entre una etiqueta y un elemento? Utilice esta entrada como referencia:

```
<title>HTML Page Title</title>
```

3. ¿Cuáles son las etiquetas entre las que se debe colocar un comentario?

4. Explique qué es un atributo y proporcione algunos ejemplos para la etiqueta `<meta>`.

## Ejercicios de exploración



1. Cree un documento HTML versión 5 simple con el título `My first HTML document` y solo un párrafo en el cuerpo, que contenga el texto `Hello World`. Utilice la etiqueta de párrafo `<p>` en el cuerpo.



2. Agregue el autor (`Kevin Author`) y la descripción (`This is my first HTML page.`) del documento HTML.

3. Agregue las siguientes palabras claves relacionadas con el documento HTML: `HTML`, `Example`, `Test` y `Metadata`.

4. Agregue el elemento `<meta charset="ISO-8859-1">` al encabezado del documento y cambie el texto `Hello World` a japonés (`□□□□□□□`). ¿Qué pasa? ¿Cómo puede solucionar el problema?

5. Después de volver a cambiar el texto del párrafo a `Hello World`, redirija la página HTML a `https://www.google.com` después de 30 segundos y agregue un comentario que explique esto en el encabezado del documento.

## Resumen

En esta lección aprendió:

- El papel de HTML
- El esqueleto HTML
- La sintaxis HTML (etiquetas, atributos, comentarios)
- El encabezado HTML
- Las metaetiquetas
- Cómo crear un documento HTML simple

En esta lección se discutieron los siguientes términos:

`<!DOCTYPE html>`
: La etiqueta de declaración.

`<html>`
: El contenedor de todas las etiquetas que componen la página HTML.

`<head>`
: El contenedor para todos los elementos de la cabecera.

`<body>`
: El contenedor de todos los elementos del cuerpo.

`<meta>`
: La etiqueta de metadatos, que se utiliza para especificar información adicional para la página HTML (como autor, descripción y codificación de caracteres).

## Respuestas a los ejercicios guiados

1. Para cada una de las siguientes etiquetas, indique la etiqueta de cierre correspondiente:

| | |
|---|---|
| `<body>` | `</body>` |
| `<head>` | `</head>` |
| `<html>` | `</html>` |
| `<meta>` | None |
| `<title>` | `</title>` |

2. ¿Cuál es la diferencia entre una etiqueta y un elemento? Utilice esta entrada como referencia:

   `<title>HTML Page Title</title>`

   Un elemento HTML consta de una etiqueta de inicio, una etiqueta de cierre y todo lo que hay entre ellas. Una etiqueta HTML se utiliza para marcar el comienzo o el final de un elemento. Por lo tanto, `<title>HTML Page Title</title>` es un elemento HTML, mientras que `<title>` y `</title>` son las etiquetas de inicio y cierre respectivamente.

3. ¿Cuáles son las etiquetas entre las que se debe colocar un comentario?

   Se inserta un comentario entre las etiquetas `<!--` y `-->` y se puede colocar en una sola línea o puede abarcar varias líneas.

4. Explique qué es un atributo y proporcione algunos ejemplos para la etiqueta `<meta>`.

   Un atributo se utiliza para especificar con mayor precisión un elemento HTML. Por ejemplo, la etiqueta `<meta>` usa el par de atributos `name` y `content` para agregar el autor y la descripción de una página HTML. En su lugar, usando el atributo `charset` puede especificar la codificación de caracteres para el documento HTML.

## Respuestas a los ejercicios de exploración

1. Cree un documento HTML simple versión 5 con el título `My first HTML document` y solo un párrafo en el cuerpo que contenga el texto `Hello World`. Use la etiqueta de párrafo `<p>` en el cuerpo.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

2. Agregue el autor (`Kevin Author`) y la descripción (`This is my first HTML page.`) del documento HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

3. Agregue las siguientes palabras clave relacionadas con el documento HTML: `HTML`, `Example`, `Test` y `Metadata`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

4. Agregue el elemento <meta charset="ISO-8859-1"> al encabezado del documento y cambie el texto Hello World a japonés (▯▯▯▯▯▯▯). ¿Qué pasa? ¿Cómo puede solucionar el problema?

   Si el ejemplo se lleva a cabo como se describe, el texto en japonés no se muestra correctamente. Esto se debe a que ISO-8859-1 representa la codificación de caracteres del alfabeto latino. Para ver el texto, necesita cambiar la codificación de caracteres, usando por ejemplo UTF-8 (<meta charset="utf-8">).

5. Después de cambiar el texto del párrafo a Hello World, redirija la página HTML a https://www.google.com después de 30 segundos y agregue un comentario que explique esto en el encabezado del documento.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
    <meta charset="utf-8">
    <!-- The page is redirected to Google after 30 seconds -->
    <meta http-equiv="refresh" content="30; url=https://www.google.com">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```




# 032.2 Semántica HTML y jerarquía de documentos

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 032.2

**Peso**

2

**Áreas de conocimiento clave**

- Crear marcado para contenidos en un documento HTML
- Comprender la estructura jerárquica del texto HTML
- Diferenciar entre elementos HTML en bloque y en línea
- Comprender elementos HTML estructurales semánticos importantes

**Lista parcial de archivos, términos y utilidades**

- `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
- `<p>`
- `<ul>`, `<ol>`, `<li>`
- `<dl>`, `<dt>`, `<dd>`
- `<pre>`
- `<strong>`, `<em>`, `<code>`
- `<b>`, `<i>`, `<tt>`
- `<span>`
- `<div>`

- `<main>`, `<header>`, `<nav>`, `<section>`, `<footer>`




# 032.2 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 032 Marcado de documentos HTML |
| **Objetivo:** | 032.2 Semántica HTML y jerarquía de documentos |
| **Lección:** | 1 de 1 |

## Introducción

En la lección anterior, aprendimos que HTML es un lenguaje de marcado que puede describir semánticamente el contenido de un sitio web. Un documento HTML contiene un llamado esqueleto que consta de los elementos HTML `<html>`, `<head>` y `<body>`. Mientras que el elemento `<head>` describe un bloque de metainformación para el documento HTML que será invisible para el visitante del sitio web, el elemento `<body>` puede contener muchos otros elementos para definir la estructura y el contenido del HTML documento.

En esta lección, veremos el formato de texto, los elementos HTML semánticos fundamentales y su propósito, y cómo estructurar un documento HTML. Usaremos una lista de compras como nuestro ejemplo.

| | |
|---|---|
| **NOTE** | Todos los ejemplos de código posteriores se encuentran dentro del elemento `<body>` de un documento HTML que contiene el esqueleto completo. Para facilitar la lectura, no mostraremos el esqueleto HTML en todos los ejemplos de esta lección. |

## Texto

En HTML, ningún bloque de texto debe estar descubierto, fuera de un elemento. Incluso un párrafo corto debe estar dentro de las etiquetas HTML `<p>`, que es el nombre corto de *paragraph*.

```
<p>Short text element spanning only one line.</p>
<p>A text element containing much longer text that may span across multiple lines,
depending on the size of the web browser window.</p>
```

Abierto en un navegador web, este código HTML produce el resultado que se muestra en Figure 1.

Short text element spanning only one line

A text element containing much longer text that may span across multiple lines depending on the size of the web browser window.

*Figure 1. Representación del navegador web de código HTML que muestra dos párrafos de texto. El primer párrafo es muy breve. El segundo párrafo es un poco más largo y se ajusta a una segunda línea.*

De forma predeterminada, los navegadores web agregan espacios antes y después de los elementos `<p>` para mejorar la legibilidad. Por esta razón, `<p>` se denomina *elemento de bloque*.

## Encabezados

HTML define seis niveles de encabezados para describir y estructurar el contenido de un documento HTML. Estos encabezados están marcados por las etiquetas HTML `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` y `<h6>`.

```
<h1>Heading level 1 to uniquely identify the page</h1>
<h2>Heading level 2</h2>
<h3>Heading level 3</h3>
<h4>Heading level 4</h4>
<h5>Heading level 5</h5>
<h6>Heading level 6</h6>
```

Un navegador web muestra este código HTML como se muestra en Figure 2.

**Headline level 1 to uniquely identify the page**

**Headline level 2**

**Headline level 3**

**Headline level 4**

**Headline level 5**

**Headline level 6**

*Figure 2. Representación del navegador web de código HTML que muestra diferentes niveles de encabezados en un documento HTML. La jerarquía de títulos se indica mediante el tamaño del texto.*

Si está familiarizado con procesadores de texto como LibreOffice o Microsoft Word, es posible que observe algunas similitudes en la forma en que un documento HTML utiliza diferentes niveles de encabezados y cómo se representan en el navegador web. De forma predeterminada, HTML usa el tamaño para indicar la jerarquía y la importancia de los encabezados y agrega espacio antes y después de cada encabezado para separarlo visualmente del contenido.

Un encabezado que usa el elemento `<h1>` está en la parte superior de la jerarquía y, por lo tanto, se considera el encabezado más importante que identifica el contenido de la página. Es comparable al elemento `<title>` discutido en la lección anterior, pero dentro del contenido del documento HTML. Los elementos de encabezado posteriores se pueden utilizar para estructurar aún más el contenido. Asegúrese de no omitir los niveles de encabezado intermedios. La jerarquía del documento debe comenzar con `<h1>`, continuar con `<h2>`, luego con `<h3>` y así sucesivamente. No es necesario utilizar todos los elementos de encabezado hasta `<h6>` si su contenido no lo exige.

**NOTE**

Los encabezados son herramientas importantes para estructurar un documento HTML, tanto semántica como visualmente. Sin embargo, los encabezados nunca deben usarse para aumentar el tamaño de texto que de otro modo no tendría importancia estructural. Por el mismo principio, no se debe poner un párrafo corto en negrita o cursiva para que parezca un encabezado; use etiquetas de encabezado para marcar los encabezados.

Comencemos a crear el documento HTML de la lista de compras definiendo su esquema. Crearemos un elemento `<h1>` para contener el título de la página, en este caso `Garden Party`, seguido de

información breve dentro de un elemento <p>. Además, usamos dos elementos <h2> para presentar las dos secciones de contenido Agenda y Please bring.

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<h2>Please bring</h2>
```

Abierto en un navegador web, este código HTML produce el resultado que se muestra en Figure 3.

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

*Figure 3. Representación del navegador web de código HTML que muestra un documento de ejemplo simple que describe una invitación a una fiesta en el jardín, con dos títulos para la agenda y una lista de cosas para llevar.*

## Saltos de línea

A veces puede ser necesario provocar un *salto de línea* sin insertar otro elemento <p> o cualquier elemento de bloque similar. En tales casos, puede utilizar el elemento de cierre automático <br>. Tenga en cuenta que debe usarse solo para insertar saltos de línea que pertenezcan al contenido, como es el caso de poemas, letras de canciones o direcciones. Si el contenido está separado por significado, es mejor usar un elemento <p> en su lugar.

Por ejemplo, podríamos dividir el texto en el párrafo de información de nuestro ejemplo anterior de la siguiente manera:

```
<p>
  Invitation to John's garden party.<br>
  Saturday, next week.
</p>
```

En un navegador web, este código HTML produce el resultado que se muestra en Figure 4.

Invitation to John's garden party.
Saturday, next week.

*Figure 4. Representación del navegador web de código HTML que muestra un documento de ejemplo simple con un salto de línea forzado.*

## Líneas horizontales

El elemento `<hr>` define una línea horizontal, también llamada *regla horizontal*. De forma predeterminada, abarca todo el ancho de su elemento padre. El elemento `<hr>` puede ayudarlo a definir un cambio temático en el contenido o separar las secciones del documento. El elemento es de cierre automático y, por lo tanto, no tiene etiqueta de cierre.

Para nuestro ejemplo, podríamos separar los dos encabezados:

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<hr>
<h2>Please bring</h2>
```

Figure 5 muestra el resultado de este código.

# Garden Party

Invitation to John's garden party on Saturday next week.

## Agenda

---

## Please bring

*Figure 5. Representación del navegador web un documento de ejemplo simple que describe una lista de compras con dos secciones separadas por una línea horizontal.*

## Listas HTML

En HTML, puede definir tres tipos de listas:

**Listas ordenadas**
: donde importa el orden de los elementos

**Listas desordenadas**
: donde el orden de los elementos no es particularmente importante

**Listas de descripción**
: para describir más de cerca algunos términos

Cada uno contiene cualquier número de *elementos de la lista*. A continuación, describiremos cada tipo de lista.

### Listas ordenadas

Una *lista ordenada* en HTML, indica mediante el elemento HTML `<ol>` una colección de cualquier número de *elementos*. Lo que hace que este elemento sea especial es que el orden de los elementos de su lista es relevante. Para enfatizar esto, los navegadores web muestran números antes de los elementos de la lista secundaria de forma predeterminada.

> **NOTE** Los elementos `<li>` son los únicos elementos secundarios válidos dentro de un elemento `<ol>`.

Para nuestro ejemplo, podemos completar la agenda de la fiesta en el jardín usando un elemento `<ol>` con el siguiente código:

```
<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

En un navegador web, este código HTML produce el resultado que se muestra en Figure 6.

## Agenda

1. Welcome
2. Barbecue
3. Dessert
4. Fireworks

*Figure 6. Representación del navegador web de un documento de ejemplo simple que contiene un título de segundo nivel seguido de una lista ordenada con cuatro elementos que describen la agenda de una fiesta en el jardín.*

### Opciones

Como puede ver en este ejemplo, los elementos de la lista están numerados con números decimales que comienzan en 1 de forma predeterminada. Sin embargo, puede cambiar este comportamiento especificando el atributo `type` de la etiqueta `<ol>`. Los valores válidos para este atributo son `1` para números decimales, `A` para letras mayúsculas, `a` para letras minúsculas, `I` para números romanos en mayúsculas e `i` para números romanos en minúsculas.

Si lo desea, también puede definir el valor inicial utilizando el atributo `start` de la etiqueta `<ol>`. El atributo `start` siempre toma un valor numérico decimal, incluso si el atributo `type` establece un tipo diferente de numeración.

Por ejemplo, podríamos ajustar la lista ordenada del ejemplo anterior para que los elementos tengan como prefijo letras mayúsculas, comenzando con la letra C, como se muestra en el siguiente ejemplo:

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Dentro de un navegador web, este código HTML se representa como Figure 7.

## Agenda

C. Welcome
D. Barbecue
E. Dessert
F. Fireworks

*Figure 7. Representación del navegador web un documento de ejemplo simple que contiene un encabezado de segundo nivel seguido de una lista ordenada con elementos precedidos por letras mayúsculas que comienzan con la letra C.*

El orden de los elementos de la lista también se puede invertir utilizando el atributo `reversed` sin un valor.

> **NOTE** En una lista ordenada, también puede establecer el valor inicial de un elemento de lista específico utilizando el atributo `value` de la etiqueta `<li>`. Los elementos de la lista que siguen se incrementarán a partir de ese número. El atributo `value` siempre toma un valor numérico decimal.

### Listas desordenadas

Una *lista desordenada* contiene una serie de elementos que, a diferencia de los de una lista ordenada, no tienen un orden o secuencia especial. El elemento HTML de esta lista es `<ul>`. Una vez más, `<li>` es el elemento HTML para marcar sus elementos de lista.

> **NOTE** Los elementos `<li>` son los únicos elementos secundarios válidos dentro de un elemento `<ul>`.

Para nuestro sitio web de ejemplo, podemos usar la lista desordenada para enumerar los elementos que los invitados pueden traer a la fiesta. Podemos lograr esto con el siguiente código HTML:

```
<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
  <li>Desserts</li>
</ul>
```

Dentro de un navegador web, este código HTML produce la pantalla que se muestra en Figure 8.

**Please bring**

- Salad
- Drinks
- Bread
- Snacks
- Desserts

*Figure 8. Representación del navegador web de un documento simple que contiene un título de segundo nivel seguido de una lista desordenada con elementos relacionados con los alimentos que se les pide a los invitados que traigan a la fiesta en el jardín.*

De forma predeterminada, cada elemento de la lista se representa mediante una viñeta de disco. Puede cambiar su apariencia usando CSS, que se discutirá en lecciones posteriores.

## Listas anidadas

Las listas se pueden anidar dentro de otras listas, como listas ordenadas dentro de listas desordenadas y viceversa. Para lograr esto, la lista anidada debe ser parte de un elemento de lista `<li>`, porque `<li>` es el único elemento hijo válido de listas ordenadas y desordenadas. Al anidar, tenga cuidado de no superponer las etiquetas HTML.

Para nuestro ejemplo, podríamos agregar alguna información de la agenda que creamos antes, como se muestra en el siguiente ejemplo:

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li>Vegetables</li>
      <li>Meat</li>
      <li>Burgers, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Un navegador web muestra el código como se muestra en Figure 9.

**Agenda**

C. Welcome
D. Barbecue
  - Vegetables
  - Meat
  - Burgers, including vegetarian options
E. Dessert
F. Fireworks

*Figure 9. Representación del navegador web de código HTML que muestra listas desordenadas anidadas dentro de una lista ordenada, para representar la agenda de una fiesta en el jardín.*

Podría ir aún más lejos y anidar varios niveles de profundidad. En teoría, no hay límite para la cantidad de listas que puede anidar. Sin embargo, al hacer esto, considere la legibilidad para sus visitantes.

### Listas de descripción

Una *lista de descripción* se define mediante el elemento `<dl>` y representa un diccionario de *claves* y *valores*. La clave es un término o nombre que desea describir y el valor es la descripción. Las listas de descripción pueden variar desde simples pares clave-valor hasta definiciones extensas.

Una clave (o *término*) se define usando el elemento `<dt>`, mientras que su descripción se define

usando el elemento <dd>.

Un ejemplo de una lista de descripción de este tipo podría ser una lista de frutas exóticas cómo muestra el ejemplo.

```
<h3>Exotic Fruits</h3>
<dl>
  <dt>Banana</dt>
  <dd>
    A long, curved fruit that is yellow-skinned when ripe. The fruit's skin
    may also have a soft green color when underripe and get brown spots when
    overripe.
  </dd>

  <dt>Kiwi</dt>
  <dd>
    A small, oval fruit with green flesh, black seeds, and a brown, hairy
    skin.
  </dd>

  <dt>Mango</dt>
  <dd>
    A fruit larger than a fist, with a green skin, orange flesh, and one big
    seed. The skin may have spots ranging from green to yellow or red.
  </dd>
</dl>
```

En un navegador web, esto produce el resultado que se muestra en Figure 10.

### Exotic Fruits

Banana
: A long, curved fruit that is yellow-skinned when ripe. The fruit's skin may also have a soft green color when underripe and get brown spots when overripe.

Kiwi
: A small, oval fruit with green flesh, black seeds and a brown, hairy skin.

Mango
: A fruit larger than a fist, with a green skin, orange flesh, and one big seed. The skin may have spots ranging from green to yellow or red.

*Figure 10. Un ejemplo de una lista de descripción HTML que utiliza frutas exóticas. La lista describe la aparición de tres frutas exóticas diferentes.*

**NOTE** A diferencia de las listas ordenadas y las listas desordenadas, en una lista de descripción, cualquier elemento HTML es válido como hijo directo. Esto le permite agrupar elementos y aplicarles estilo en otro lugar utilizando CSS.

## Formato de texto en línea

En HTML, puede utilizar elementos de formato para cambiar la apariencia del texto. Estos elementos se pueden categorizar como *elementos de presentación* o *elementos de frase*.

### Elementos de presentación

Los elementos de presentación básicos cambian la fuente o el aspecto del texto; estos son `<b>`, `<i>`, `<u>` y `<tt>`. Estos elementos se definieron originalmente antes de que CSS permitiera convertir el texto en negrita, cursiva, etc. Ahora, por lo general, existen mejores formas de alterar el aspecto del texto, pero a veces aún se ven estos elementos.

## Texto en negrita

Para colocar el texto en negrita, puede insertarlo dentro del elemento `<b>` como se ilustra en el siguiente ejemplo. El resultado aparece en Figure 11.

```
This <b>word</b> is bold.
```

This **word** is bold.

*Figure 11. La etiqueta `<b>` se usa para poner el texto en negrita.*

De acuerdo con la especificación HTML5, el elemento `<b>` debe usarse solo cuando no haya más etiquetas apropiadas. El elemento que produce el mismo resultado visual, pero además agrega importancia semántica al texto marcado, es `<strong>`.

## Texto en cursiva

Para colocar el texto en cursiva, puede insertarlo dentro del elemento `<i>` como se ilustra en el siguiente ejemplo. El resultado aparece en Figure 12.

```
This <i>word</i> is in italics.
```

This *word* is in italics.

*Figure 12. La etiqueta* <i> *se usa para poner el texto en cursiva.*

De acuerdo con la especificación HTML 5, el elemento <i> debe usarse solo cuando no haya etiquetas más apropiadas.

## Texto subrayado

Para subrayar el texto, puede insertarlo dentro del elemento <u> como se ilustra en el siguiente ejemplo. El resultado aparece en Figure 13.

```
This <u>word</u> is underlined.
```

This word is underlined.

*Figure 13. La etiqueta* <u> *se utiliza para subrayar el texto.*

De acuerdo con la especificación HTML 5, el elemento <u> debe usarse solo cuando no haya mejores formas de subrayar el texto. CSS proporciona una alternativa moderna.

## Fuente de ancho fijo o monoespaciada

Para mostrar texto en una fuente monoespaciada (ancho fijo), que a menudo se usa para mostrar código de computadora, puede usar el elemento <tt> como se ilustra en el siguiente ejemplo. El resultado aparece en Figure 14.

```
This <tt>word</tt> is in fixed-width font.
```

This word is in fixed-width font.

*Figure 14. La etiqueta* <tt> *se usa para mostrar texto en una fuente de ancho fijo.*

La etiqueta <tt> no es compatible con HTML5. Los navegadores web aún lo procesan como se esperaba. Sin embargo, debería utilizar etiquetas más apropiadas, que incluyen <code>, <kbd>, <var> y <samp>.

### Elementos frase

Los elementos de frase no solo cambian la apariencia del texto, sino que también agregan

importancia semántica a una palabra o frase. Utilizándolos, puede enfatizar una palabra o marcarla como importante. Estos elementos, a diferencia de los elementos de presentación, son reconocidos por los lectores de pantalla, lo que hace que el texto sea más accesible para los visitantes con discapacidad visual y permite que los motores de búsqueda lean y evalúen mejor el contenido de la página. Los elementos de frase que usaremos a lo largo de esta lección son <em>, <strong> y <code>.

## Texto enfatizado

Para enfatizar el texto, puede insertarlo dentro del elemento <em> como se ilustra en el siguiente ejemplo:

```
This <em>word</em> is emphasized.
```

This *word* is emphasized.

*Figure 15. La etiqueta <em> se usa para enfatizar el texto.*

Como puede ver, los navegadores web muestran <em> de la misma manera que <i>, pero <em> agrega importancia semántica como un elemento de frase, lo que mejora la accesibilidad para los visitantes con discapacidad visual.

## Texto importante

Para marcar el texto como importante, puede insertarlo dentro del elemento <strong> como se ilustra en el siguiente ejemplo. El resultado se muestra en Figure 16.

```
This <strong>word</strong> is important.
```

This **word** is important.

*Figure 16. La etiqueta <strong> se utiliza para marcar el texto como importante.*

Como se visualiza, los navegadores web muestran <strong> de la misma manera que <b>, pero <strong> agrega importancia semántica como un elemento de frase, lo que mejora la accesibilidad para los visitantes con discapacidad visual.

## Código de computadora

Para insertar un fragmento de código de computadora, puede insertarlo dentro del elemento <code> como se ilustra en el siguiente ejemplo. El resultado se muestra en Figure 17.

```
The Markdown code <code># Heading</code> creates a heading at the highest level in
the hierarchy.
```

The Markdown code `# Heading` creates a heading at the highest level in the hierarchy.

*Figure 17. La etiqueta `<code>` se usa para insertar un fragmento de código de computadora.*

## Texto resaltado

Para resaltar texto con un fondo amarillo, similar al estilo de un resaltador, puede usar el elemento `<mark>` como se ilustra en el siguiente ejemplo. El resultado se muestra en Figure 18.

```
This <mark>word</mark> is highlighted.
```

This word is highlighted.

*Figure 18. La etiqueta `<mark>` se utiliza para resaltar el texto con un fondo amarillo.*

### Dar formato al texto de nuestra lista de compras HTML

Basándonos en nuestros ejemplos anteriores, insertemos algunos elementos de frase para cambiar la apariencia del texto y al mismo tiempo agregar importancia semántica. El resultado aparece en Figure 19.

```
<h1>Garden Party</h1>
<p>
  Invitation to <strong>John's garden party</strong>.<br>
  <strong>Saturday, next week.</strong>
</p>

<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li><em>Vegetables</em></li>
      <li><em>Meat</em></li>
      <li><em>Burgers</em>, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li><mark>Fireworks</mark></li>
</ol>

<hr>

<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
  <li>Desserts</li>
</ul>
```

# Garden Party

Invitation to **John's garden party**.
**Saturday, next week.**

## Agenda

1. Welcome
2. Barbecue
   - *Vegetables*
   - *Meat*
   - *Burgers*, including vegetarian options
3. Dessert
4. Fireworks

---

## Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

*Figure 19. La página HTML con algunos elementos de formato.*

En este documento HTML de muestra, la información más importante sobre la fiesta en el jardín en sí se marca como importante mediante el elemento <strong>. Los alimentos que están disponibles para la barbacoa se enfatizan mediante el elemento <em>. Los fuegos artificiales simplemente se resaltan usando el elemento <mark>.

Como ejercicio, puede intentar formatear otros fragmentos de texto utilizando también los otros elementos de formato.

### Texto preformateado

En la mayoría de los elementos HTML, los espacios en blanco generalmente se reducen a un solo espacio o incluso se ignoran por completo. Sin embargo, hay un elemento HTML llamado <pre> que le permite definir el llamado texto *preformateado*. El espacio en blanco en el contenido de este

elemento, incluidos los espacios y los saltos de línea, se conservan y representan en el navegador web. Además, el texto se muestra en una fuente de ancho fijo, similar al elemento `<code>`.

```
<pre>
field() {
  shift $1 ; echo $1
}
</pre>
```

```
field() {
  shift $1 ; echo $1
}
```

*Figure 20. Representación del código HTML del navegador web que ilustra cómo el elemento HTML `<pre>` conserva los espacios en blanco.*

## Agrupar elementos

Por convención, los elementos HTML se dividen en dos categorías:

**Elementos en bloque**

Aparecen en una nueva línea y ocupan todo el ancho disponible. Ejemplos de elementos de bloque que ya discutimos son `<p>`, `<ol>` y `<h2>`.

**Elementos en línea**

Estos aparecen en la misma línea que otros elementos y texto, ocupando solo el espacio que requiera su contenido. Ejemplos de elementos de línea son `<strong>`, `<em>` y `<i>`.

> **NOTE** HTML5 ha introducido categorías de elementos más precisas y precisas, tratando de evitar confusiones con el bloque CSS y los cuadros en línea. Para simplificar, nos enfocaremos aquí a la subdivisión convencional de elementos en bloque y en línea.

Los elementos fundamentales para agrupar varios son los elementos `<div>` y `<span>`.

El elemento `<div>` es un contenedor en bloque para otros elementos HTML y no agrega valor semántico por sí mismo. Puede utilizar este elemento para dividir un documento HTML en secciones y estructurar su contenido, tanto para la legibilidad del código como para aplicar estilos CSS a un grupo de elementos, como aprenderá en una lección posterior.

De forma predeterminada, los navegadores web siempre insertan un salto de línea antes y después de cada elemento <div> para que cada uno se muestre en su propia línea.

Por el contrario, el elemento <span> se usa como contenedor para texto HTML y generalmente se usa para agrupar otros elementos en línea con el fin de aplicar estilos usando CSS a una porción más pequeña de texto.

El elemento <span> se comporta como un texto normal y no comienza en una nueva línea. Por tanto, es un elemento en línea.

El siguiente ejemplo compara la representación visual del elemento semántico <p> y los elementos de agrupación <div> y <span>:

```
<p>Text within a paragraph</p>
<p>Another paragraph of text</p>
<hr>
<div>Text wrapped within a <code>div</code> element</div>
<div>Another <code>div</code> element with more text</div>
<hr>
<span>Span content</span>
<span>and more span content</span>
```

Un navegador web muestra este código como en Figure 21.

Text within a paragraph

Another paragraph of text

---

Text wrapped within a div element
Another div element with more text

---

Span content and more span content

*Figure 21. Representación del navegador web de un documento de prueba que ilustra las diferencias entre los elementos de párrafo, div y span en HTML.*

Ya vimos que, de forma predeterminada, el navegador web agrega espacios antes y después de los elementos <p>. Este espaciado no se aplica a ninguno de los elementos de agrupación <div> y <span>. Sin embargo, los elementos <div> se formatean como sus propios bloques, mientras que el texto de los elementos <span> se muestra en la misma línea.

## Estructura de la página HTML

Hemos discutido cómo usar elementos HTML para describir semánticamente el contenido de una página web, en otras palabras, para transmitir significado y contexto. Otro grupo de elementos está diseñado con el propósito de describir la *estructura semántica* de una página web, una expresión o su estructura. Estos elementos son elementos en bloque, es decir, visualmente se comportan de manera similar a un elemento `<div>`. Su propósito es definir la estructura semántica de una página web especificando áreas bien definidas como encabezados, pies de página y el contenido principal de la página. Estos elementos permiten la agrupación semántica de contenidos para que también puedan ser entendidos por un ordenador, incluidos los motores de búsqueda y los lectores de pantalla.

### El elemento `<header>`

El elemento `<header>` contiene información introductoria al elemento semántico circundante dentro de un documento HTML. (`<h1>`, …, `<h6>`) tienen diferentes usos, sin embargo un `<header>` a menudo incluye un elemento de encabezado (`<h1>`, …, `<h6>`).

En la práctica, este elemento se utiliza con mayor frecuencia para representar el encabezado de la página, como un banner con un logotipo. También se puede utilizar para introducir el contenido de cualquiera de los siguientes elementos: `<body>`, `<section>`, `<article>`, `<nav>`, o `<aside>`.

Un documento puede tener varios elementos `<header>`, pero un elemento `<header>` no se puede anidar dentro de otro elemento `<header>`. Tampoco se puede utilizar un elemento `<footer>` dentro de un `<header>`.

Por ejemplo, para agregar un encabezado de página a nuestro documento de ejemplo, podemos hacer lo siguiente:

```
<header>
  <h1>Garden Party</h1>
</header>
```

No habrá cambios visibles en el documento HTML, ya que `<h1>` (como todos los demás elementos de encabezado) es un elemento a nivel de bloque sin más propiedades visuales.

### El elemento de contenido `<main>`

El elemento `<main>` es un contenedor para el contenido central de una página web. No debe haber más de un elemento `<main>` en un documento HTML.

En nuestro documento de ejemplo, todo el código HTML que hemos escrito hasta ahora se colocaría

dentro del elemento `<main>`.

```
<main>
  <header>
    <h1>Garden Party</h1>
  </header>
  <p>
    Invitation to <strong>John's garden party</strong>.<br>
    <strong>Saturday, next week.</strong>
  </p>

  <h2>Agenda</h2>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>

  <hr>

  <h2>Please bring</h2>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</main>
```

Como el elemento `<header>`, el elemento `<main>` no causa ningún cambio visual en nuestro ejemplo.

### El elemento `<footer>`

El elemento `<footer>` contiene notas a pie de página, por ejemplo, información de autoría,

información de contacto o documentos relacionados, para el elemento semántico que lo rodea, p. Ej. `<section>`, `<nav>`, o `<aside>`. Un documento puede tener varios elementos `<footer>` que le permiten describir mejor los elementos semánticos. Sin embargo, un elemento `<footer>` no se puede anidar dentro de otro elemento `<footer>`, ni se puede usar un elemento `<header>` dentro de un `<footer>`.

Para nuestro ejemplo, podemos agregar información de contacto para el anfitrión (John) como se muestra en el siguiente ejemplo:

```
<footer>
  <p>John Doe</p>
  <p>john.doe@example.com</p>
</footer>
```

#### El elemento `<nav>`

El elemento `<nav>` describe una unidad de navegación principal, como un menú, que contiene una serie de hipervínculos.

| NOTE | No todos los hipervínculos deben incluirse dentro de un elemento `<nav>`. Es útil cuando se listan un grupo de enlaces. |
|---|---|

Debido a que los hipervínculos aún no se han cubierto, el elemento de navegación no se incluirá en el ejemplo de esta lección.

#### El elemento `<aside>`

El elemento `<aside>` es un contenedor que no es necesario dentro del orden del contenido de la página principal, pero que generalmente está relacionado indirectamente o es complementario. Este elemento se utiliza a menudo para las barras laterales que muestran información secundaria, como un glosario.

Para nuestro ejemplo, podemos agregar información de dirección y viaje, que solo están indirectamente relacionados con el contenido restante, usando el elemento `<aside>`.

```
<aside>
  <p>
    10, Main Street<br>
    Newville
  </p>
  <p>Parking spaces available.</p>
</aside>
```

### El elemento `<section>`

El elemento `<section>` define una sección lógica en un documento que es parte del elemento semántico circundante, pero que no funcionaría como contenido independiente, como un capítulo.

En nuestro documento de ejemplo, podemos ajustar las secciones de contenido para la agenda y traer secciones de lista como se muestra en el siguiente ejemplo:

```
<section>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</section>

<hr>

<section>
  <header>
    <h2>Please bring</h2>
  </header>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</section>
```

Este ejemplo también agrega más elementos `<header>` dentro de las secciones, de modo que cada sección está dentro de su propio elemento `<header>`.

### El elemento `<article>`

El elemento `<article>` define contenido independiente y autónomo que tiene sentido por sí solo sin el resto de la página. Su contenido es potencialmente redistribuible o reutilizable en otro contexto. Los ejemplos típicos o el material apropiado para un elemento `<article>` son una publicación en un blog, una lista de productos para una tienda y un anuncio de un producto. El anuncio podría existir

tanto por sí solo como dentro de una página más grande.

En nuestro ejemplo, podemos reemplazar la primera `<section>` que envuelve la agenda con un elemento `<article>`.

```
<article>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</article>
```

El elemento `<header>` que agregamos en el ejemplo anterior también puede persistir aquí, porque los elementos `<article>` pueden tener sus propios elementos `<header>`.

### El ejemplo final

Combinando todos los ejemplos anteriores, el documento HTML final de nuestra invitación tiene el siguiente aspecto:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Garden Party</title>
  </head>

  <body>
    <main>
      <h1>Garden Party</h1>
      <p>
```

```
      Invitation to <strong>John's garden party</strong>.<br>
      <strong>Saturday, next week.</strong>
    </p>

    <article>
      <h2>Agenda</h2>
      <ol>
        <li>Welcome</li>
        <li>
          Barbecue
          <ul>
            <li><em>Vegetables</em></li>
            <li><em>Meat</em></li>
            <li><em>Burgers</em>, including vegetarian options</li>
          </ul>
        </li>
        <li>Dessert</li>
        <li><mark>Fireworks</mark></li>
      </ol>
    </article>

    <hr>

    <section>
      <h2>Please bring</h2>
      <ul>
        <li>Salad</li>
        <li>Drinks</li>
        <li>Bread</li>
        <li>Snacks</li>
        <li>Desserts</li>
      </ul>
    </section>
  </main>

  <aside>
    <p>
      10, Main Street<br>
      Newville
    </p>
    <p>Parking spaces available.</p>
  </aside>

  <footer>
    <p>John Doe</p>
```

```
      <p>john.doe@example.com</p>
    </footer>
  </body>
</html>
```

En un navegador web, toda la página se representa como se muestra en Figure 22.

# Garden Party

Invitation to **John's garden party**.
**Saturday, next week.**

## Agenda

1. Welcome
2. Barbecue
   - *Vegetables*
   - *Meat*
   - *Burgers*, including vegetarian options
3. Dessert
4. Fireworks

---

## Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

10, Main Street
Newville

Parking spaces available.

John Doe

john.doe@example.com

*Figure 22. Representación del navegador web del documento HTML resultante que combina los ejemplos anteriores. La página representa una invitación a una fiesta en el jardín y describe la agenda de la noche y una lista de comida para que los invitados lleven.*

## Ejercicios guiados

1. Para cada una de las siguientes etiquetas, indique la etiqueta de cierre correspondiente:

| | |
|---|---|
| `<h5>` | |
| `<br>` | |
| `<ol>` | |
| `<dd>` | |
| `<hr>` | |
| `<strong>` | |
| `<tt>` | |
| `<main>` | |

2. Para cada una de las siguientes etiquetas, indique si marca el comienzo de un bloque o elemento en línea:

| | |
|---|---|
| `<h3>` | |
| `<span>` | |
| `<b>` | |
| `<div>` | |
| `<em>` | |
| `<dl>` | |
| `<li>` | |
| `<nav>` | |
| `<code>` | |
| `<pre>` | |

3. ¿Qué tipo de listas se pueden crear en HTML? ¿Qué etiquetas debería utilizar para cada una de ellas?

| |
|---|
| |

4. ¿Qué etiquetas encierran los elementos en bloque que puede utilizar para estructurar una página HTML?

| |
|---|
| |

## Ejercicios de exploración

1. Cree una página HTML básica con el título “Form Rules”. Utilizará esta página HTML para todos los ejercicios de exploración, cada uno de los cuales se basa en los anteriores. Luego agregue un encabezado de nivel 1 con el texto “How to fill in the request form”, un párrafo con el texto “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” y una lista desordenada con los siguientes elementos: “Name”, “Surname”, “Email Address”, “Nation”, “Country”, y “Zip/Postal Code”.

2. Ponga los primeros tres campos (“Name”, “Surname”, y “Email Address”) en negrita, a la vez que agrega importancia semántica. Luego agregue un encabezado de nivel 2 con el texto “Required fields” y un párrafo con el texto “Bold fields are mandatory.”

3. Agregue otro encabezado de nivel 2 con el texto “Steps to follow”, un párrafo con el texto “There are four steps to follow:”, y una lista ordenada con los siguientes elementos: “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive” y “Check your e-mail - You will receive the full HTML course in minutes”.

4. Usando `<div>`, cree un bloque para cada sección que comience con un encabezado de nivel 2.

5. Usando `<div>`, cree otro bloque para la sección comenzando con el título de nivel 1. Luego divida esta sección de las otras dos con una línea horizontal.

6. Agregue el elemento de encabezado con el texto “Form Rules - 2021” y el elemento de pie de página con el texto “Copyright Note - 2021”. Finalmente, agregue el elemento principal que debe contener los tres bloques `<div>`.

## Resumen

En esta lección aprendió:

- Cómo crear el marcado para contenidos en un documento HTML
- La estructura jerárquica de texto HTML
- La diferencia entre elementos HTML en bloque y en línea
- Cómo crear documentos HTML con estructura semántica

En esta lección se discutieron los siguientes términos:

`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
: Las etiquetas de encabezado.

`<p>`
: La etiqueta de párrafo.

`<ol>`
: La etiqueta de lista ordenada.

`<ul>`
: La etiqueta de lista desordenada.

`<li>`
: La etiqueta del elemento de la lista.

`<dl>`
: La etiqueta de la lista de descripción.

`<dt>`, `<dd>`
: Las etiquetas de cada término y descripción para una lista de descripción.

`<pre>`
: La etiqueta de conservación de formato.

`<b>`, `<i>`, `<u>`, `<tt>`, `<em>`, `<strong>`, `<code>`, `<mark>`
: Las etiquetas de formato.

**`<div>`, `<span>`**

Las etiquetas de agrupación.

**`<header>`, `<main>`, `<nav>`, `<aside>`, `<footer>`**

Las etiquetas utilizadas para proporcionar una estructura y un diseño simples a una página HTML.

## Respuestas a los ejercicios guiados

1. Para cada una de las siguientes etiquetas, indique la etiqueta de cierre correspondiente:

| | |
|---|---|
| `<h5>` | `</h5>` |
| `<br>` | Does not exist |
| `<ol>` | `</ol>` |
| `<dd>` | `</dd>` |
| `<hr>` | Does not exist |
| `<strong>` | `</strong>` |
| `<tt>` | `</tt>` |
| `<main>` | `</main>` |

2. Para cada una de las siguientes etiquetas, indique si marca el comienzo de un bloque o elemento en línea:

| | |
|---|---|
| `<h3>` | Block Element |
| `<span>` | Inline Element |
| `<b>` | Inline Element |
| `<div>` | Block Element |
| `<em>` | Inline Element |
| `<dl>` | Block Element |
| `<li>` | Block Element |
| `<nav>` | Block Element |
| `<code>` | Inline Element |
| `<pre>` | Block Element |

3. ¿Qué tipo de listas se pueden crear en HTML? ¿Qué etiquetas debería utilizar para cada una de ellas?

   En HTML, puede crear tres tipos de listas: listas ordenadas que consisten en una serie de elementos de lista numerados, listas desordenadas que consisten en una serie de elementos de lista que no tienen un orden o secuencia especial y listas de descripción que representan entradas como en un diccionario o enciclopedia. Una lista ordenada se encierra entre las etiquetas `<ol>` y `</ol>`, una lista desordenada se encierra entre las etiquetas `<ul>` y `</ul>`, y una

lista de descripción se encierra entre las etiquetas `<dl>` y `</dl>`. Cada elemento de una lista ordenada o desordenada se incluye entre las etiquetas `<li>` y `</li>`, mientras que cada término de una lista de descripción se incluye entre las etiquetas `<dt>` y `</dt>` y su descripción se incluye entre las etiquetas `<dd>` y `</dd>`.

4. ¿Qué etiquetas encierran los elementos de bloque que puede usar para estructurar una página HTML?

   Las etiquetas `<header>` y `</header>` encierran el encabezado de la página, las etiquetas `<main>` y `</main>` encierran el contenido principal de la página HTML, las etiquetas `<nav>` y `</nav>` encierran la llamada barra de navegación, las etiquetas `<aside>` y `</aside>` encierran la barra lateral, y las etiquetas `<footer>` y `</footer>` encierran la página pie de página.

## Respuestas a los ejercicios de exploración

1. Cree una página HTML básica con el título “Form Rules”. Utilizará esta página HTML para todos los ejercicios de exploración, cada uno de los cuales se basa en los anteriores. Luego agregue un encabezado de nivel 1 con el texto “How to fill in the request form”, un párrafo con el texto “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” y una lista desordenada con los siguientes elementos de la lista: “Name”, “Surname”, “Email Address”, “Nation”, “Country”, y “Zip/Postal Code”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
necessary
      to fill in the following fields:
    </p>
    <ul>
      <li>Name</li>
      <li>Surname</li>
      <li>Email Address</li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>
  </body>
</html>
```

2. Ponga los primeros tres campos (“Name”, “Surname”, y “Email Address”) en negrita, al mismo tiempo que agrega importancia semántica. Luego agregue un encabezado de nivel 2 con el texto “Required fields” y un párrafo con el texto “Bold fields are mandatory.”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>
  </body>
</html>
```

3. Agregue otro encabezado de nivel 2 con el texto “Steps to follow”, un párrafo con el texto “There are four steps to follow:”, y una lista ordenada con los siguientes elementos de la lista: “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, y “Check your e-mail - You will receive the full HTML course in minutes”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>

    <h2>Steps to follow</h2>
    <p>There are four steps to follow:</p>
    <ol>
      <li>Fill in the fields</li>
      <li>Click the Submit button</li>
      <li>
        Check your e-mail and confirm your request by clicking on the link you
        receive
      </li>
      <li>
        Check your e-mail - You will receive the full HTML course in minutes
      </li>
    </ol>
  </body>
</html>
```

4. Usando `<div>`, cree un bloque para cada sección que comience con un encabezado de nivel 2.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
you
          receive
        </li>
        <li>
          Check your e-mail - You will receive the full HTML course in minutes
        </li>
      </ol>
    </div>
  </body>
</html>
```

5. Usando <div>, cree otro bloque para la sección que comienza con el encabezado de nivel 1. Luego divide esta sección de las otras dos con una línea horizontal.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <div>
      <h1>How to fill in the request form</h1>
      <p>
        To receive the PDF document with the complete HTML course, it is
        necessary to fill in the following fields:
      </p>
      <ul>
        <li><strong> Name </strong></li>
        <li><strong> Surname </strong></li>
        <li><strong> Email Address </strong></li>
        <li>Nation</li>
        <li>Country</li>
        <li>Zip/Postal Code</li>
      </ul>
    </div>

    <hr>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
you
          receive
        </li>
```

```
        <li>
          Check your e-mail - You will receive the full HTML course in minutes
        </li>
      </ol>
    </div>
  </body>
</html>
```

6. Agregue el elemento de encabezado con el texto “Form Rules - 2021” y el elemento de pie de página con el texto “Copyright Note - 2021”. Finalmente, agregue el elemento principal que debe contener los tres bloques `<div>`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <header>
      <h1>Form Rules - 2021</h1>
    </header>

    <main>
      <div>
        <h1>How to fill in the request form</h1>
        <p>
          To receive the PDF document with the complete HTML course, it is
          necessary to fill in the following fields:
        </p>
        <ul>
          <li><strong> Name </strong></li>
          <li><strong> Surname </strong></li>
          <li><strong> Email Address </strong></li>
          <li>Nation</li>
          <li>Country</li>
          <li>Zip/Postal Code</li>
        </ul>
      </div>

      <hr>

      <div>
        <h2>Required fields</h2>
```

```
        <p>Bold fields are mandatory.</p>
      </div>

      <div>
        <h2>Steps to follow</h2>
        <p>There are four steps to follow:</p>
        <ol>
          <li>Fill in the fields</li>
          <li>Click the Submit button</li>
          <li>
            Check your e-mail and confirm your request by clicking on the link
            you receive
          </li>
          <li>
            Check your e-mail - You will receive the full HTML course in
minutes
          </li>
        </ol>
      </div>
    </main>

    <footer>
      <p>Copyright Note - 2021</p>
    </footer>
  </body>
</html>
```




## 032.3 Referencias HTML y recursos incrustados

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 032.3

**Peso**

2

**Áreas de conocimiento clave**

- Crear enlaces a recursos externos y anclajes de página
- Agregar imágenes a documentos HTML
- Comprender las propiedades clave de los formatos de archivo de medios comunes, incluidos PNG, JPG y SVG
- Conocimiento de iframes

**Lista parcial de archivos, términos y utilidades**

- Atributo `id`
- `<a>`, incluidos los atributos `href` y `target` (`_blank`, `_self`, `_parent`, `_top`)
- `<img>`, incluidos los atributos `src` y `alt`




# 032.3 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 032 Marcado de documentos HTML |
| **Objetivo:** | 032.3 Referencias HTML y recursos incrustados |
| **Lección:** | 1 de 1 |

## Introducción

Cualquier página web moderna rara vez está formada solo por texto. Comprende muchos otros tipos de contenidos, como imágenes, audio, video e incluso otros documentos HTML. Junto con el contenido externo, los documentos HTML pueden contener enlaces a otros documentos, lo que hace que la experiencia de navegar por Internet mucho más simple.

### Contenido incrustado

El intercambio de archivos es posible a través de Internet sin páginas web escritas en HTML, entonces, ¿por qué HTML es el formato elegido para documentos web y no PDF o cualquier otro formato de procesamiento de texto? Una razón importante es que HTML mantiene sus recursos multimedia en archivos separados. En un entorno como Internet, donde la información a menudo es redundante y se distribuye en diferentes ubicaciones, es importante evitar transferencias de datos innecesarias. La mayoría de las veces, las nuevas versiones de una página web extraen las mismas imágenes y otros archivos de soporte que las versiones anteriores, por lo que el navegador web puede utilizar los archivos recuperados anteriormente en lugar de copiar todo de nuevo. Además, mantener los archivos separados facilita la personalización del contenido multimedia de acuerdo con las características del cliente, como su ubicación, tamaño de pantalla y velocidad de conexión.

### Imágenes

El tipo más común de contenido incrustado son las imágenes que acompañan al texto. Las imágenes se guardan por separado y se hace referencia a ellas dentro del archivo HTML con la etiqueta `<img>`:

```
<img src="logo.png">
```

La etiqueta `<img>` no requiere una etiqueta de cierre. La propiedad `src` indica la ubicación de origen del archivo de imagen. En este ejemplo, el archivo de imagen `logo.png` debe estar ubicado en el mismo directorio que el archivo HTML; de lo contrario, el navegador no podrá mostrarlo. La propiedad de ubicación de origen acepta rutas relativas, por lo que la *notación de puntos* se puede utilizar para indicar la ruta a la imagen:

```
<img src="../logo.png">
```

Los dos puntos indican que la imagen se encuentra dentro del directorio principal en relación con el directorio donde está el archivo HTML. Si el nombre de archivo `../logo.png` se usa dentro de un archivo HTML cuya URL es `http://example.com/library/periodicals/index.html`, el navegador solicitará el archivo de imagen en la dirección `http://example.com/library/logo.png`.

La notación de puntos también se aplica si el archivo HTML no es un archivo real en el sistema de archivos; el navegador HTML interpreta la URL como si fuera una ruta a un archivo, pero es el trabajo del servidor HTTP decidir si esa ruta se refiere a un archivo o al contenido generado dinámicamente. El dominio y la ruta adecuada se agregan automáticamente a todas las solicitudes al servidor, en caso de que el archivo HTML provenga de una solicitud HTTP. Del mismo modo, el navegador abrirá la imagen adecuada si el archivo HTML se abrió directamente desde el sistema de archivos local.

Las ubicaciones de origen que comienzan con una barra inclinada `/` se tratan como rutas absolutas. Las rutas absolutas tienen información completa de las ubicaciones de la imagen, por lo que funcionan independientemente de la ubicación del documento HTML. Si el archivo de imagen se encuentra en otro servidor, que será el caso cuando se utilice una *Content Delivery Network (Red de entrega de contenido)* (CDN), también se debe incluir el nombre de dominio.

**NOTE** Las redes de entrega de contenido están compuestas por servidores distribuidos geográficamente que almacenan contenido estático para otros sitios web. Ayudan a mejorar el rendimiento y la disponibilidad de los sitios de mayor acceso.

Si la imagen no se puede cargar, el navegador HTML mostrará el texto proporcionado por el atributo

`alt` en lugar de la imagen. Por ejemplo:

```
<img src="logo.png" alt="The Company logo">
```

El atributo `alt` también es importante para la accesibilidad. Los navegadores de solo texto y los lectores de pantalla lo utilizan como descripción de la imagen correspondiente.

## Tipos de imágenes

Los navegadores web pueden mostrar todos los tipos de imágenes populares, como JPEG, PNG, GIF y SVG. Las dimensiones de las imágenes se detectan tan pronto como se cargan las imágenes, pero se pueden predefinir con los atributos `width` y `height`:

```
<img src="logo.png" alt="The Company logo" width="300" height="100">
```

La única razón para incluir atributos de dimensión en la etiqueta `<img>` es evitar romper el diseño cuando la imagen tarda demasiado en cargarse o cuando no se puede cargar en absoluto. El uso de los atributos `width` y `height` para cambiar las dimensiones originales de la imagen puede generar resultados no deseados:

- Las imágenes se distorsionarán cuando el tamaño original sea más pequeño que las nuevas dimensiones o cuando la nueva relación de proporción difiera del original.
- Reducir el tamaño de las imágenes grandes utiliza un ancho de banda adicional que resultará en tiempos de carga más largos.

SVG es el único formato que no sufre estos efectos, porque toda su información gráfica se almacena en coordenadas numéricas muy adecuadas para escalar y sus dimensiones no afectan el tamaño del archivo (de ahí el nombre *Scalable Vector Graphics ó Gráficos vectoriales escalables*). Por ejemplo, solo la posición, las dimensiones laterales y la información de color son necesarias para dibujar un rectángulo en SVG. El valor particular de cada píxel se renderizará dinámicamente posteriormente. De hecho, las imágenes SVG son similares a los archivos HTML, en el sentido de que sus elementos gráficos también se definen mediante etiquetas en un archivo de texto. Los archivos SVG están pensados para representar dibujos de bordes definidos, como gráficos o diagramas.

Las imágenes que no se ajusten a estos criterios deben almacenarse como *bitmaps (mapas de bits)*. A diferencia de los formatos de imagen basados en vectores, los mapas de bits almacenan información de color para cada píxel de la imagen de antemano. El almacenamiento del valor de color para cada píxel de la imagen genera una gran cantidad de datos, por lo que los mapas de bits generalmente se almacenan en formatos comprimidos, como JPEG, PNG o GIF.

Se recomienda el formato JPEG para fotografías, porque su algoritmo de compresión produce buenos resultados para sombras y fondos borrosos. Para imágenes en las que predominan los colores sólidos, el formato PNG es más apropiado. Por lo tanto, se debe elegir el formato PNG cuando sea necesario convertir una imagen vectorial en un mapa de bits.

El formato GIF ofrece la calidad de imagen más baja de todos los formatos de mapa de bits populares. Sin embargo, todavía se usa ampliamente debido a su soporte para animaciones. De hecho, muchos sitios web emplean archivos GIF para mostrar videos cortos, pero existen mejores formas de mostrar contenido de video.

### Audio y video

Los contenidos de audio y video se pueden agregar a un documento HTML más o menos de la misma manera que las imágenes. Como era de esperar, la etiqueta para agregar audio es `<audio>` y la etiqueta para agregar video es `<video>`. Obviamente, los navegadores de solo texto no pueden reproducir contenido multimedia, por lo que las etiquetas `<audio>` y `<video>` emplean la etiqueta de cierre para contener el texto utilizado como respaldo para el elemento que no se pudo mostrar. Por ejemplo:

```
<audio controls src="/media/recording.mp3">
<p>Unable to play <em>recording.mp3</em></p>
</audio>
```

Si el navegador no admite la etiqueta `<audio>`, en su lugar se mostrará la línea "Unable to play recording.mp3". El uso de etiquetas de cierre `</audio>` o `</video>` permite que una página web incluya contenido alternativo más elaborado que la simple línea de texto permitida por el atributo `alt` de la etiqueta `<img>`.

El atributo `src` para las etiquetas `<audio>` y `<video>` funciona de la misma manera que para la etiqueta `<img>`, pero también acepta URL que apunta a una transmisión en vivo. El navegador se encarga de almacenar en búfer, decodificar y mostrar el contenido a medida que se recibe. El atributo `controls` muestra los controles de reproducción. Sin él, el visitante no podrá pausar, rebobinar ni controlar la reproducción.

### Contenido genérico

Un documento HTML se puede anidar en otro documento HTML, de manera similar a la inserción de una imagen en un documento HTML, pero usando la etiqueta `<iframe>`:

```
<iframe name="viewer" src="gallery.html">
<p>Unsupported browser</p>
</iframe>
```

Los navegadores de solo texto más simples no admiten la etiqueta `<iframe>` y, en su lugar, mostrarán el texto adjunto. Al igual que con las etiquetas multimedia, el atributo `src` establece la ubicación de origen del documento anidado. Los atributos `width` y `height` se pueden agregar para cambiar las dimensiones predeterminadas del elemento `iframe`.

El atributo `name` permite hacer referencia al iframe y cambiar el documento anidado. Sin este atributo, el documento anidado no se puede cambiar. Se puede usar un elemento `anchor` para cargar un documento desde otra ubicación dentro de un iframe en lugar de la ventana actual del navegador.

## Enlaces

El elemento de la página comúnmente denominado *link (enlace web)* también se conoce con el término técnico *anchor*, de ahí el uso de la etiqueta `<a>`. El anchor conduce a otra ubicación, que puede ser cualquier dirección admitida por el navegador. La ubicación está indicada por el atributo `href` (*hyperlink reference*):

```
<a href="contact.html">Contact Information</a>
```

La ubicación se puede escribir como una ruta relativa o absoluta, como con los contenidos incrustados discutidos anteriormente. Solo el contenido de texto adjunto (por ejemplo, `Contact Information`) es visible para el visitante, generalmente como texto azul subrayado en el que se puede hacer clic de forma predeterminada, pero el elemento que se muestra sobre el enlace también puede ser cualquier otro contenido visible, como imágenes:

```
<a href="contact.html"><img src="contact.png" alt="Contact Information"></a>
```

Se pueden agregar prefijos especiales a la ubicación para indicarle al navegador cómo abrirlo. Si el anchor apunta a una dirección de correo electrónico, por ejemplo, su atributo `href` debe incluir el prefijo `mailto:`:

```
<a href="mailto:info@lpi.org">Contact by email</a>
```

El prefijo `tel:` indica un número de teléfono. Es particularmente útil para los visitantes que ven la página en dispositivos móviles:

```
<a href="tel:+123456789">Contact by phone</a>
```

Cuando se hace clic en el enlace, el navegador abre el contenido de la ubicación con la aplicación asociada.

El uso más común del anchor es cargar otros documentos web. De forma predeterminada, el navegador reemplazará el documento HTML actual con contenido en la nueva ubicación. Este comportamiento se puede modificar utilizando el atributo `target`. El objetivo `_blank`, por ejemplo, le dice al navegador que abra la ubicación dada en una nueva ventana o nueva pestaña del navegador, dependiendo de las preferencias del visitante:

```
<a href="contact.html" target="_blank">Contact Information</a>
```

El objetivo `_self` es el predeterminado cuando no se proporciona el atributo `target`. Hace que el documento referenciado reemplace al documento actual.

Otros tipos de objetivos están relacionados con el elemento `<iframe>`. Para cargar un documento referenciado dentro de un elemento `<iframe>`, el atributo `target` debe apuntar al nombre del elemento iframe:

```
<p><a href="gallery.html" target="viewer">Photo Gallery</a></p>

<iframe name="viewer" width="800" height="600">
<p>Unsupported browser</p>
</iframe>
```

El elemento iframe funciona como una ventana de navegador distinta, por lo que cualquier enlace cargado desde el documento dentro del iframe reemplazará solo el contenido del iframe. Para cambiar ese comportamiento, los elementos de anclaje dentro del documento enmarcado también pueden usar el atributo `target`. El destino `_parent`, cuando se usa dentro de un documento enmarcado, hará que la ubicación referenciada reemplace el documento principal que contiene la etiqueta `<iframe>`. Por ejemplo, el documento `gallery.html` incrustado podría contener un anchor que se carga a sí mismo mientras reemplaza el documento principal:

```
<p><a href="gallery.html" target="_parent">Open as parent document</a></p>
```

Los documentos HTML admiten varios niveles de anidamiento con la etiqueta `<iframe>`. El destino `_top`, cuando se usa en un anchor dentro de un documento enmarcado, hará que la ubicación

referenciada reemplace al documento principal en la ventana del navegador, independientemente de si es el padre inmediato del `<iframe>` correspondiente o un antepasado adicional de vuelta en la cadena.

### Ubicaciones dentro de los documentos

La dirección de un documento HTML puede contener opcionalmente un *fragmento* que se puede utilizar para identificar un recurso dentro del documento. Este fragmento, también conocido como *URL anchor*, es una cadena que sigue un signo de almohadilla `#` al final de la URL. Por ejemplo, la palabra `History` es el anchor en la URL `https://en.wikipedia.org/wiki/Internet#History`.

Cuando la URL tiene un ancla, el navegador se desplazará al elemento correspondiente en el documento: es decir, el elemento cuyo atributo `id` es igual al ancla en la URL. En el caso de la URL dada, `https://en.wikipedia.org/wiki/Internet#History`, el navegador saltará directamente a la sección “History”. Examinando el código HTML de la página, encontramos que el título de la sección tiene el atributo `id` correspondiente:

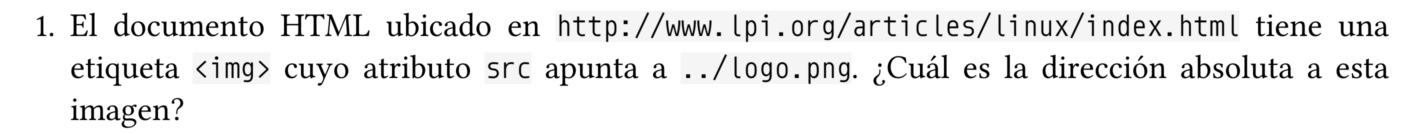
```
<span class="mw-headline" id="History">History</span>
```

Los anchor de URL se pueden usar en el atributo `href` de la etiqueta `<a>`, ya sea cuando apuntan a páginas externas o cuando apuntan a ubicaciones dentro de la página actual. En el último caso, basta con comenzar con el signo de almohadilla con el fragmento de URL, como en `<a href="#History">History</a>`.

| | |
|---|---|
| **WARNING** | El atributo `id` no debe contener espacios en blanco (espacios, tabulaciones, etc.) y debe ser único dentro del documento. |

Hay formas de personalizar cómo reaccionará el navegador a los anchor de URL. Es posible, por ejemplo, escribir una función de JavaScript que escuche el evento de ventana *hashchange* y active una acción personalizada, como una animación o una solicitud HTTP. Sin embargo, vale la pena señalar que el fragmento de URL nunca se envía al servidor con la URL, por lo que el servidor HTTP no puede usarlo como identificador.

## Ejercicios guiados



1. El documento HTML ubicado en `http://www.lpi.org/articles/linux/index.html` tiene una etiqueta `<img>` cuyo atributo `src` apunta a `../logo.png`. ¿Cuál es la dirección absoluta a esta imagen?

2. Nombra dos razones por las que el atributo `alt` es importante en las etiquetas `<img>`.

3. ¿Qué formato de imagen proporciona una buena calidad y mantiene el tamaño del archivo pequeño cuando se usa para fotografías con puntos borrosos y con muchos colores y sombras?

4. En lugar de utilizar un proveedor externo como Youtube, ¿qué etiqueta HTML le permite incrustar un archivo de video en un documento HTML utilizando solo funciones HTML estándar?

## Ejercicios de exploración

1. Suponga que un documento HTML tiene el hipervínculo `<a href="pic1.jpeg">First picture</a>` y el elemento iframe `<iframe name="gallery"></iframe>`. ¿Cómo podría modificar la etiqueta de hipervínculo para que la imagen a la que apunta se cargue dentro del elemento iframe dado después de que el usuario haga clic en el enlace?

2. ¿Qué sucederá cuando el visitante haga clic en un hipervínculo en un documento dentro de un iframe y el hipervínculo tiene el atributo de destino establecido en `_self`?

3. Observe que el anchor de URL para la segunda sección de su página HTML no funciona. ¿Cuál es la causa probable de este error?

## Resumen

Esta lección cubre cómo agregar imágenes y otro contenido multimedia usando las etiquetas HTML adecuadas. Además, el lector aprende las diferentes formas en que se pueden utilizar los hipervínculos para cargar otros documentos y señalar ubicaciones específicas dentro de una página. La lección pasa por los siguientes conceptos y procedimientos:

- La etiqueta `<img>` y sus principales atributos: `src` y `alt`.
- Rutas de URL relativas y absolutas.
- Formatos de imágenes populares para la Web y sus características.
- Las etiquetas multimedia `<audio>` y `<video>`.
- Cómo insertar documentos anidados con la etiqueta `<iframe>`.
- La etiqueta de hipervínculo `<a>`, su atributo `href` y objetivos especiales.
- Cómo utilizar fragmentos de URL, también conocidos como anclajes hash.

## Respuestas a los ejercicios guiados

1. El documento HTML ubicado en `http://www.lpi.org/articles/linux/index.html` tiene una etiqueta `<img>` cuyo atributo `src` apunta a `../logo.png`. ¿Cuál es la dirección absoluta a esta imagen?

   `http://www.lpi.org/articles/logo.png`

2. Nombra dos razones por las que el atributo `alt` es importante en las etiquetas `<img>`.

   Los navegadores de solo texto podrán mostrar una descripción de la imagen que falta. Los lectores de pantalla utilizan el atributo `alt` para describir la imagen.

3. ¿Qué formato de imagen proporciona una buena calidad y mantiene el tamaño del archivo pequeño cuando se usa para fotografías con puntos borrosos y con muchos colores y sombras?

   El formato JPEG.

4. En lugar de utilizar un proveedor externo como Youtube, ¿qué etiqueta HTML le permite incrustar un archivo de video en un documento HTML utilizando solo funciones HTML estándar?

   La etiqueta `<video>`.

## Respuestas a los ejercicios de exploración

1. Suponga que un documento HTML tiene el hipervínculo `<a href="pic1.jpeg">First picture</a>` y el elemento iframe `<iframe name="gallery"></iframe>`. ¿Cómo podría modificar la etiqueta de hipervínculo para que la imagen a la que apunta se cargue dentro del elemento iframe dado después de que el usuario haga clic en el enlace?

   Usando el atributo `target` de la etiqueta `a`: `<a href="pic1.jpeg" target="gallery">First picture</a>`.

2. ¿Qué sucederá cuando el visitante haga clic en un hipervínculo en un documento dentro de un iframe y el hipervínculo tiene el atributo de destino establecido en `_self`?

   El documento se cargará dentro del mismo iframe, que es el comportamiento predeterminado.

3. Observe que el anchor de URL para la segunda sección de su página HTML no funciona. ¿Cuál es la causa probable de este error?

   El fragmento de URL después del signo de almohadilla no coincide con el atributo `id` en el elemento correspondiente a la segunda sección, o el atributo `id` del elemento no está presente.




# 032.4 Formularios HTML

### Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 032.4

### Peso

2

### Áreas de conocimiento clave

- Crear formularios HTML simples
- Comprender los métodos de formulario HTML
- Comprender los elementos y tipos de entrada HTML

### Lista parcial de archivos, términos y utilidades

- `<form>`, incluidos los atributos de `method` (`get`, `post`), `action` y `enctype`
- `<input>`, ncluido el atributo `type` (`text`, `email`, `password`, `number`, `date`, `file`, `range`, `radio`, `checkbox`, `hidden`)
- `<button>`, incluido el atributo `type` (`submit`, `reset`, `hidden`)
- `<textarea>`
- Atributos de elementos de formulario comunes (`name`, `value`, `id`)
- `<label>`, incluido el atributo `for`




# 032.4 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 032 Marcado de documentos HTML |
| **Objetivo:** | 032.4 Formularios HTML |
| **Lección:** | 1 de 1 |

## Introducción

Los formularios web proporcionan una forma sencilla y eficaz de solicitar información de los visitantes desde una página HTML. El desarrollador de front-end puede utilizar varios componentes, como campos de texto, casillas de verificación, botones y muchos otros para crear interfaces que enviarán datos al servidor de forma estructurada.

### Formularios HTML simples

Antes de saltar al código de marcado específico de los formularios, comencemos con un documento HTML simple en blanco, sin ningún contenido de cuerpo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- The body content goes here -->

</body>
</html>
```

Guarde el ejemplo de código como un archivo de texto sin formato con una extensión `.html` (por ejemplo `form.html`) y use su navegador favorito para abrirlo. Después de modificar el código, presione el botón de recarga en el navegador para mostrar las modificaciones.

La estructura básica del formulario viene dada por la propia etiqueta `<form>` y sus elementos internos:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- Form to collect personal information -->

<form>

<h2>Personal Information</h2>

<p>Full name:</p>
<p><input type="text" name="fullname" id="fullname"></p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>

</body>
</html>
```

Las comillas dobles no son necesarias para atributos de una sola palabra como `type`, por lo que el resultado no varía al usuar `type=text` o `type="text"`. El desarrollador puede elegir qué convención utilizar.

Guarde el nuevo contenido y vuelva a cargar la página en el navegador. Debería ver el resultado que se muestra en Figure 23.




*Figure 23. Un formulario muy básico.*

La etiqueta `<form>` por sí sola no produce ningún resultado notable en la página. Los elementos dentro de las etiquetas `<form>…</form>` definirán los campos y otras ayudas visuales que se muestran al visitante.

El código de ejemplo contiene etiquetas HTML generales (`<h2>` y `<p>`) y la etiqueta `<input>`, que es una etiqueta específica de formulario. Mientras que las etiquetas generales pueden aparecer en cualquier parte del documento, las etiquetas específicas del formulario deben usarse solo dentro del elemento `<form>`; es decir, entre las etiquetas de apertura `<form>` y de cierre `</form>`.

> **NOTE** HTML proporciona solo etiquetas y propiedades básicas para modificar la apariencia estándar de los formularios. CSS proporciona mecanismos elaborados para modificar la apariencia del formulario, por lo que la recomendación es escribir código HTML que trate solo con los aspectos funcionales del formulario y modificar su apariencia con CSS.

Como se muestra en el ejemplo, la etiqueta de párrafo `<p>` se puede utilizar para describir el campo al visitante. Sin embargo, no hay una forma obvia de que el navegador pueda relacionar la descripción en la etiqueta `<p>` con el elemento de entrada correspondiente. La etiqueta `<label>` es más apropiada en estos casos (de ahora en adelante, considere que todos los ejemplos de código están dentro del cuerpo del documento HTML):

```
<form>

<h2>Personal Information</h2>

<label for="fullname">Full name:</label>
<p><input type="text" name="fullname" id="fullname"></p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>
```

El atributo `for` en la etiqueta `<label>` contiene el `id` del elemento de entrada correspondiente. Hace que la página sea más accesible, ya que los lectores de pantalla podrán decir el contenido del elemento de etiqueta cuando el de entrada esté enfocado. Además, los visitantes pueden hacer clic en la etiqueta para colocar el foco en su campo de entrada correspondiente.

El atributo `id` funciona para elementos de formulario como lo hace para cualquier otro elemento en el documento. Proporciona un identificador para el elemento que es único en todo el documento. El atributo `name` tiene un propósito similar, pero se usa para identificar el elemento de entrada en el contexto del formulario. El navegador usa el atributo `name` para identificar el campo de entrada cuando envía los datos del formulario al servidor, por lo que es importante asignar valores significativos y únicos al atributo `name` dentro del formulario.

`type` es el atributo principal del elemento de entrada, porque controla el tipo de datos que acepta el elemento y su presentación visual para el visitante. Si no se proporciona el atributo `type`, de forma predeterminada, la entrada muestra un cuadro de texto. Los siguientes tipos de entrada son compatibles con los navegadores modernos:

*Table 1. Tipos de entrada de formulario*

| Tipo de atributo | Tipo de dato | Cómo se muestra |
|---|---|---|
| `hidden` | Una cadena arbitraria | N/A |
| `text` | Texto sin saltos de línea | Un control de texto |
| `search` | Texto sin saltos de línea | Un control de búsqueda |
| `tel` | Texto sin saltos de línea | Un control de texto |
| `url` | Una URL absoluta | Un control de texto |

| Tipo de atributo | Tipo de dato | Cómo se muestra |
|---|---|---|
| `email` | Una dirección de correo electrónico o una lista de direcciones de correo electrónico | Un control de texto |
| `password` | Texto sin saltos de línea (información sensible) | Un control de texto que oculta la entrada de datos |
| `date` | Una fecha (año, mes, día) sin zona horaria | Un control de fecha |
| `month` | Una fecha que consta de un año y un mes sin zona horaria | Control de un mes |
| `week` | Una fecha que consta de un número de semana-año y un número de semana sin zona horaria | Control de una semana |
| `time` | Una hora (hora, minuto, segundo, fracción de segundo) sin zona horaria | un control de tiempo |
| `datetime-local` | Una fecha y hora (año, mes, día, hora, minuto, segundo, fracción de segundo) sin zona horaria | Un control de fecha y hora |
| `number` | Un valor numérico | Un control de texto o un control giratorio |
| `range` | Un valor numérico, con el extra semántico de que el valor exacto no es importante | Un control deslizante o similar |
| `color` | Un color sRGB con componentes rojo, verde y azul de 8 bits | Un selector de color |
| `checkbox` | Un conjunto de cero o más valores de una lista predefinida | Una casilla de verificación (ofrece opciones y permite seleccionar múltiples opciones) |
| `radio` | Un valor enumerado | Un botón de radio (ofrece opciones y permite seleccionar solo una opción) |

| Tipo de atributo | Tipo de dato | Cómo se muestra |
|---|---|---|
| `file` | Cero o más archivos, cada uno con un tipo MIME y un nombre de archivo opcional | Una etiqueta y un botón |
| `submit` | Un valor enumerado, que finaliza el proceso de entrada y hace que se envíe el formulario | Un botón |
| `image` | Una coordenada, relativa al tamaño de una imagen en particular, que finaliza el proceso de entrada y hace que se envíe el formulario | Una imagen en la que se puede hacer clic o un botón |
| `button` | N/A | Un botón genérico |
| `reset` | N/A | Un botón cuya función es restablecer todos los demás campos a sus valores iniciales |

La apariencia de los tipos de entrada `password`, `search`, `tel`, `url`, e `email` no difieren del tipo estándar de `text`. Su propósito es ofrecer sugerencias al navegador sobre el contenido previsto para ese campo de entrada, por lo que el navegador o el script que se ejecuta en el lado del cliente puede realizar acciones personalizadas para un tipo de entrada específico. La única diferencia entre el tipo de entrada de texto y el de contraseña, por ejemplo, es que el contenido del campo de contraseña no se muestra cuando el visitante los escribe. En dispositivos de pantalla táctil, donde el texto se escribe con un teclado, el navegador solo puede mostrar el teclado numérico cuando una entrada de tipo `tel` gana el foco. Otra acción posible es sugerir una lista de direcciones de correo electrónico conocidas cuando se trata de una entrada de tipo `email`.

El tipo `number` también aparece como una entrada de texto simple, pero con flechas de incremento/decremento a un lado. Su uso hará que el teclado numérico aparezca en los dispositivos con pantalla táctil cuando tenga el foco.

Los otros elementos de entrada tienen su propia apariencia y comportamiento. El tipo `date`, por ejemplo, se representa de acuerdo con la configuración del formato de fecha local y se muestra un calendario cuando el campo gana el foco:

```
<form>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date">
</p>

</form>
```

Figure 24 muestra cómo la versión de escritorio de Firefox representa actualmente este campo.

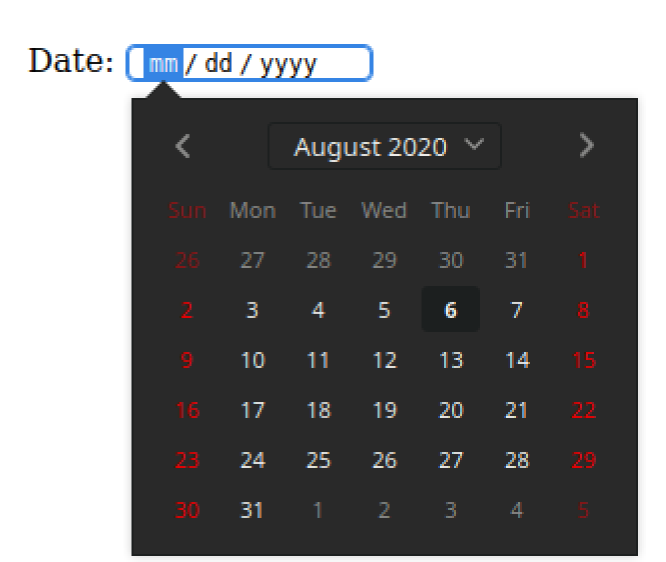



*Figure 24. El tipo de entrada de fecha.*

**NOTE** Los elementos pueden aparecer ligeramente diferentes en diferentes navegadores o sistemas operativos, pero su funcionamiento y uso son siempre los mismos.

Esta es una característica estándar en todos los navegadores modernos y no requiere opciones o programación adicionales.

Independientemente del tipo de entrada, el contenido de un campo de entrada se denomina *value*. Todos los valores de campo están vacíos de forma predeterminada, pero el atributo `value` se puede utilizar para establecer un valor predeterminado para el campo. El valor del tipo de fecha debe utilizar el formato *YYYY-MM-DD*. El valor predeterminado en el siguiente campo de fecha es el 6 de agosto de 2020:

```
<form>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date" value="2020-08-06">
</p>

</form>
```

Los tipos de entrada específicos ayudan al visitante a completar los campos, pero no le impiden eludir las restricciones e ingresar valores arbitrarios. Por eso es importante que los valores de los campos se validen cuando lleguen al servidor.

Los elementos de formulario cuyos valores debe escribir el visitante pueden tener atributos especiales que ayuden a completarlos. El atributo `placeholder` inserta un valor de ejemplo en el elemento de entrada:

```
<p>Address: <input type="text" name="address" id="address" placeholder="e.g. 41
John St., Upper Suite 1"></p>
```

El texto de ejemplo aparece dentro del elemento de entrada, como se muestra en Figure 25.

Address: e.g. 41 John St., Upper Suite 1

*Figure 25. Ejemplo de atributo de* `placeholder`.

Una vez que el visitante comienza a escribir en el campo, el texto de ejemplo desaparece y no se envía como valor de campo si el visitante deja el campo vacío.

El atributo `required` requiere que el visitante inserte un valor para el campo correspondiente antes de enviar el formulario:

```
<p>Address: <input type="text" name="address" id="address" required
placeholder="e.g. 41 John St., Upper Suite 1"></p>
```

`required` es un atributo booleano, por lo que puede colocarse solo (sin el signo igual). Es importante marcar los campos que son obligatorios, de lo contrario, el visitante no podrá saber qué campos faltan y evitar el envío del formulario.

El atributo `autocomplete` indica si el navegador puede completar automáticamente el valor del elemento de entrada. Si se establece en `autocomplete="off"`, entonces el navegador no sugiere valores pasados para completar la entrada. Los elementos de entrada para información confidencial, como números de tarjetas de crédito, deben tener el atributo `autocomplete` establecido en `off`.

## Entrada para textos grandes: textarea

El elemento `textarea` permite al visitante ingresar más de una línea de texto, a diferencia del campo de texto que solo permite una. El área de texto es un elemento separado, pero no se basa en el elemento de entrada:

```
<p> <label for="comment">Type your comment here:</label> <br>

<textarea id="comment" name="comment" rows="10" cols="50">
My multi-line, plain-text comment.
</textarea>

</p>
```

La apariencia típica de un `textarea` es Figure 26.

*Figure 26. El elemento* `textarea`.

A diferencia de otros elementos de entrada `textarea` tiene una etiqueta de cierre (`</textarea>`), por lo que su contenido (es decir, su valor) va entre ellos. Los atributos `rows` y `cols` no limitan la cantidad

de texto; se utilizan solo para definir el diseño. El área de texto también tiene un control en la esquina inferior derecha, lo que permite al visitante cambiar su tamaño.

## Listas de opciones

Se pueden usar varios tipos de controles de formulario para presentar una lista de opciones al visitante: el elemento `<select>` y los tipos de entrada `radio` y `checkbox`.

El elemento `<select>` es un control desplegable con una lista de entradas predefinidas:

```
<p><label for="browser">Favorite Browser:</label>
<select name="browser" id="browser">
<option value="firefox">Mozilla Firefox</option>
<option value="chrome">Google Chrome</option>
<option value="opera">Opera</option>
<option value="edge">Microsoft Edge</option>
</select>
</p>
```

La etiqueta `<option>` representa una sola entrada en el control `<select>` correspondiente. La lista completa aparece cuando el visitante toca o hace clic en el control, como se muestra en Figure 27.

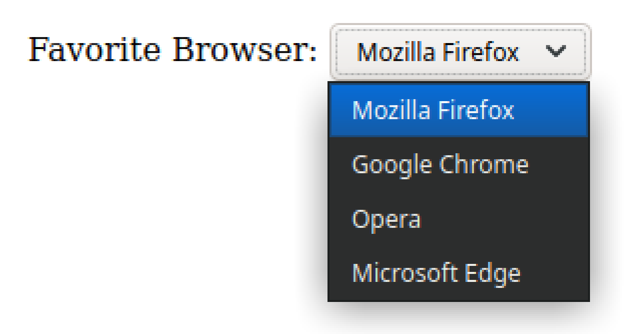



*Figure 27. El elemento de formulario* `select`.

La primera entrada de la lista está seleccionada de forma predeterminada. Para cambiar este comportamiento, puede agregar el atributo `selected` a otra entrada para que se seleccione cuando se cargue la página.

El tipo de entrada `radio` es similar al control `<select>`, pero en lugar de una lista desplegable, muestra todas las entradas para que el visitante pueda marcar una de ellas. Los resultados del siguiente código se muestran en Figure 28.

```
<p>Favorite Browser:</p>

<p>
    <input type="radio" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="radio" id="browser-chrome" name="browser" value="chrome">
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="radio" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="radio" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>
```

Favorite Browser:

Mozilla Firefox

Google Chrome

Opera

Microsoft Edge

*Figure 28. Elementos de entrada de tipo* `radio`*.*

Tenga en cuenta que todos los tipos de entrada `radio` del mismo grupo tienen el mismo atributo `name`. Cada uno de ellos es exclusivo, por lo que el atributo `value` correspondiente para la entrada elegida será el asociado con el atributo `name` compartido. `checked` funciona como el atributo `selected` del control `<select>`. Marca la entrada correspondiente cuando la página se carga por primera vez. La etiqueta `<label>` es especialmente útil para las entradas de radio, ya que permite al visitante comprobar una entrada haciendo clic o tocando el texto correspondiente además del control en sí.

Mientras que los controles de `radio` están pensados para seleccionar una única entrada de una lista, el tipo de entrada `checkbox` permite al visitante seleccionar varias:

```
<p>Favorite Browser:</p>

<p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
    <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
    <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
    <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
    <label for="browser-edge">Microsoft Edge</label>
</p>
```

Las casillas de verificación también pueden usar el atributo `checked` para hacer que las entradas se seleccionen de forma predeterminada. En lugar de los controles redondos de la entrada de radio, las casillas de verificación se representan como controles cuadrados, como se muestra en Figure 29.

Favorite Browser:

☑ Mozilla Firefox

☑ Google Chrome

☐ Opera

☐ Microsoft Edge

*Figure 29. El tipo de entrada* `checkbox`.

Si se selecciona más de una entrada, el navegador las enviará con el mismo nombre, lo que requiere que el desarrollador de backend escriba un código específico para leer correctamente los datos del formulario que contienen casillas de verificación.

Para mejorar la usabilidad, los campos de entrada se pueden agrupar dentro de una etiqueta `<fieldset>`:

```
<fieldset>
<legend>Favorite Browser</legend>

<p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
    <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
    <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
    <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
    <label for="browser-edge">Microsoft Edge</label>
</p>
</fieldset>
```

La etiqueta `<legend>` contiene el texto que se coloca en la parte superior del marco que la etiqueta `<fieldset>` dibuja alrededor de los controles (Figure 30).

Favorite Browser

- [x] Mozilla Firefox
- [x] Google Chrome
- [ ] Opera
- [ ] Microsoft Edge

*Figure 30. Agrupación de elementos con la etiqueta `fieldset`.*

La etiqueta `<fieldset>` no cambia la forma en que se envían los valores de campo al servidor, pero permite al desarrollador de la interfaz manipular los controles anidados más fácilmente. Por ejemplo, establecer `disabled` en un atributo `<fieldset>` hará que todos sus elementos internos no estén disponibles para el visitante.

## El tipo de elemento `hidden`

Hay situaciones en las que el desarrollador quiere incluir información en el formulario que el visitante no puede manipular, es decir, enviar un valor elegido por el desarrollador sin presentar un campo de formulario donde el visitante puede escribir o cambiar el valor. El desarrollador puede querer, por ejemplo, incluir una ficha de identificación para ese formulario en particular que no necesita ser visto por el visitante. Un elemento de formulario oculto se codifica como en el siguiente ejemplo:

```
<input type="hidden" id="form-token" name="form-token" value="e730a375-b953-4393-847d-2dab065bbc92">
```

El valor de un campo de entrada oculto generalmente se agrega al documento en el lado del servidor, cuando se procesa el documento. Las entradas ocultas se tratan como campos ordinarios cuando el navegador las envía al servidor, que también las lee como campos de entrada ordinarios.

## El tipo de entrada file

Además de los datos textuales, ya sea ingresados o seleccionados de una lista, los formularios HTML también pueden enviar archivos arbitrarios al servidor. El tipo de entrada `file` permite al visitante elegir un archivo del sistema de archivos local y enviarlo directamente desde la página web:

```
<p>
<label for="attachment">Attachment:</label><br>
<input type="file" id="attachment" name="attachment">
</p>
```

En lugar de un campo de formulario para escribir o seleccionar un valor, el tipo de entrada `file` muestra un botón `browse` que abrirá un cuadro de diálogo de archivo. Cualquier tipo de archivo es aceptado por el tipo de entrada `file`, pero el desarrollador de backend probablemente restringirá los tipos de archivos permitidos y su tamaño máximo. La verificación del tipo de archivo también se puede realizar en la interfaz agregando el atributo `accept`. Para aceptar solo imágenes JPEG y PNG, por ejemplo, el atributo `accept` debe ser `accept="image/jpeg, image/png"`.

## Botones de acción

De forma predeterminada, el formulario se envía cuando el visitante presiona la tecla Intro en cualquier campo de entrada. Para hacer las cosas más intuitivas, se debe agregar un botón de envío con el tipo de entrada `submit`:

```
<input type="submit" value="Submit form">
```

El texto del atributo `value` se muestra en el botón, como en Figure 31.

*Figure 31. Un botón de envío estándar.*

Otro botón útil para incluir en formularios complejos es el botón `reset`, que borra el formulario y lo devuelve a su estado original:

```
<input type="reset" value="Clear form">
```

Al igual que el botón de envío, el texto del atributo `value` se utiliza para etiquetar el botón. Alternativamente, la etiqueta `<button>` se puede usar para agregar botones a formularios o en cualquier otro lugar de la página. A diferencia de los botones hechos con la etiqueta `<input>`, el elemento del botón tiene una etiqueta de cierre y el texto entre ellas es su contenido interno:

```
<button>Submit form</button>
```

Cuando está dentro de un formulario, la acción predeterminada para el elemento `button` es enviar el formulario. Al igual que los botones de entrada, el atributo de tipo del botón se puede cambiar a `reset`.

## Acciones y métodos de los formularios

El último paso para escribir un formulario HTML es definir cómo y adónde se deben enviar los datos. Estos aspectos dependen de los detalles tanto del cliente como del servidor.

En el lado del servidor, el enfoque más común es tener un archivo de secuencia de comandos que analizará, validará y procesará los datos del formulario de acuerdo con el propósito de la aplicación. Por ejemplo, el desarrollador de backend podría escribir un script llamado `receive_form.php` para recibir los datos enviados desde el formulario. En el lado del cliente, el script se indica en el atributo `action` de la etiqueta del formulario:

```
<form action="receive_form.php">
```

El atributo `action` sigue las mismas convenciones que todas las direcciones HTTP. Si el script está en el mismo nivel jerárquico de la página que contiene el formulario, se puede escribir sin su ruta. De lo contrario, se debe proporcionar la ruta absoluta o relativa. El script también debe generar la respuesta para que sirva como página de destino, cargada por el navegador después de que el visitante envía el formulario.

HTTP proporciona distintos métodos para enviar datos de formularios a través de una conexión con el servidor. Los métodos más comunes son `get` y `post`, que deben indicarse en el atributo `method` de la etiqueta `form`:

```
<form action="receive_form.php" method="get">
```

O:

```
<form action="receive_form.php" method="post">
```

En el método `get`, los datos del formulario se codifican directamente en la URL de la solicitud. Cuando el visitante envía el formulario, el navegador cargará la URL definida en el atributo `action` con los campos del formulario adjuntos.

Se prefiere el método `get` para pequeñas cantidades de datos, como formularios de contacto simples. Sin embargo, la URL no puede exceder algunos miles de caracteres, por lo que el método `post` debe

usarse cuando los formularios contienen campos grandes o no textuales, como imágenes.

El método `post` hace que el navegador envíe los datos del formulario en la sección del cuerpo de la solicitud HTTP. Si bien es necesario para datos binarios que exceden el límite de tamaño de una URL, el método `post` agrega una sobrecarga innecesaria a la conexión cuando se usa en formularios textuales más simples, por lo que se prefiere el método `get` en tales casos.

El método elegido no afecta la forma en que el visitante interactúa con el formulario. Los métodos `get` y `post` son procesados de manera diferente por el script del lado del servidor que recibe el formulario.

Cuando se utiliza el método `post`, también es posible cambiar el tipo MIME del contenido del formulario con el atributo `enctype`. Esto afecta la forma en que los campos de formulario y los valores se apilarán juntos en la comunicación HTTP con el servidor. El valor predeterminado para `enctype` es `application/x-www-form-urlencoded`, que es similar al formato utilizado en el método `get`. Si el formulario contiene campos de entrada de tipo `file`, en su lugar debe usarse el enctype `multipart/form-data`.

## Ejercicios guiados

1. ¿Cuál es la forma correcta de asociar una etiqueta `<label>` a una etiqueta `<input>`?

2. ¿Qué tipo de elemento de entrada proporciona un control deslizante para elegir un valor numérico?

3. ¿Cuál es el propósito del atributo de formulario `placeholder`?

4. ¿Cómo podría hacer que la segunda opción de un control de selección esté seleccionada por defecto?

## Ejercicios de exploración

1. ¿Cómo pudiera cambiar el comportamiento de un control de entrada de archivos para que solo acepte archivos PDF?

2. ¿Cómo podría informar al visitante qué campos de un formulario son obligatorios?

3. Reúna tres fragmentos de código en esta lección en un solo formulario. Asegúrese de no utilizar el mismo atributo `name` o `id` en varios controles de formulario.

## Resumen

Esta lección cubre cómo crear formularios HTML simples para enviar datos al servidor. En el lado del cliente, los formularios HTML constan de elementos HTML estándares que se combinan para crear interfaces personalizadas. Además, los formularios deben configurarse para comunicarse correctamente con el servidor. La lección abarca los siguientes conceptos y procedimientos:

- La etiqueta `<form>` y la estructura básica del formulario.
- Elementos de entrada básicos y especiales.
- La función de etiquetas especiales como `<label>`, `<fieldset>` y `<legend>`.
- Botones y acciones de formulario.

## Respuestas a los ejercicios guiados

1. ¿Cuál es la forma correcta de asociar una etiqueta `<label>` a una etiqueta `<input>`?

   El atributo `for` de la etiqueta `<label>` debe contener la identificación de la etiqueta `<input>` correspondiente.

2. ¿Qué tipo de elemento de entrada proporciona un control deslizante para elegir un valor numérico?

   El tipo de entrada `range`.

3. ¿Cuál es el propósito del atributo de formulario `placeholder`?

   El atributo `placeholder` contiene un ejemplo de entrada para el visitante que es visible cuando el campo de entrada correspondiente está vacío.

4. ¿Cómo podría hacer que la segunda opción de un control de selección esté seleccionada por defecto?

   El segundo elemento `option` debe tener el atributo `selected`.

## Respuestas a los ejercicios de exploración

1. ¿Cómo pudiera cambiar el comportamiento de un control de entrada de archivos para que solo acepte archivos PDF?

   El atributo `accept` del elemento de entrada debe establecerse en `application/pdf`.

2. ¿Cómo podría informar al visitante qué campos de un formulario son obligatorios?

   Por lo general, los campos obligatorios están marcados con un asterisco (`*`), y una nota breve como "Los campos marcados con * son obligatorios" se coloca cerca del formulario.

3. Reúna tres fragmentos de código en esta lección en un solo formulario. Asegúrese de no utilizar el mismo atributo `name` o `id` en varios controles de formulario.

```
<form action="receive_form.php" method="get">

<h2>Personal Information</h2>

<label for="fullname">Full name:</label>
<p><input type="text" name="fullname" id="fullname"></p>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date">
</p>

<p>Favorite Browser:</p>

<p>
  <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="checkbox" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
  <input type="checkbox" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>
```



# Tema 033: Estilo de contenido CSS




# 033.1 Conceptos básicos de CSS

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 033.1

**Peso**

1

**Áreas de conocimiento clave**

- Incrustar CSS en un documento HTML
- Comprender la sintaxis de CSS
- Agregar comentarios a CSS
- Conocimiento de las funciones y los requisitos de accesibilidad

**Lista parcial de archivos, términos y utilidades**

- Atributos de estilo y tipo HTML (text/css)
- `<style>`
- `<link>`, incluidos los atributos `rel` (stylesheet), `type` (text/css) y `src`
- `;`
- `/,/`




# 033.1 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 033 Estilo de contenido CSS |
| **Objetivo:** | 033.1 Conceptos básicos de CSS |
| **Lección:** | 1 de 1 |

## Introducción

Todos los navegadores web renderizan páginas HTML utilizando reglas de presentación predeterminadas que son prácticas y sencillas, pero no visualmente atractivas. HTML por sí solo ofrece pocas características para escribir páginas elaboradas basadas en conceptos modernos de experiencia de usuario. Después de escribir páginas HTML simples, probablemente se habrá dado cuenta de que son antiestéticas en comparación con las páginas bien diseñadas que se encuentran en Internet. Esto se debe a que, en HTML moderno, el código de marcado destinado a la estructura y función de los elementos del documento (es decir, el *contenido semántico*) está separado de las reglas que definen cómo deben verse los elementos (la *presentación*). Las reglas de presentación están escritas en un lenguaje diferente llamado *Cascading Style Sheets* (CSS). Permite cambiar casi todos los aspectos visuales del documento, como las fuentes, los colores y la ubicación de los elementos a lo largo de la página.

En la mayoría de los casos, los documentos HTML no están pensados para mostrarse en un diseño fijo como un archivo PDF. Más bien, las páginas web basadas en HTML probablemente se renderizarán en una amplia variedad de tamaños de pantalla o incluso se imprimirán. CSS proporciona opciones ajustables para garantizar que la página web se procese según lo previsto, ajustada al dispositivo o aplicación que la abre.

## Aplicar estilos

Hay varias formas de incluir CSS en un documento HTML: insertando el código directamente en la etiqueta del elemento, en una sección separada del código fuente de la página o en un archivo externo para ser referenciado por la página HTML.

### El atributo `style`

La forma más sencilla de modificar el estilo de un elemento específico es escribirlo directamente en la etiqueta del elemento usando el atributo `style`. Todos los elementos HTML visibles permiten un atributo `style`, cuyo valor puede ser una o más reglas CSS, también conocidas como *properties*. Comencemos con un ejemplo simple, un elemento de párrafo:

```
<p>My stylized paragraph</p>
```

La sintaxis básica de una propiedad CSS personalizada es `property: value`, donde `property` es el aspecto particular que desea cambiar en el elemento y `value` define el reemplazo de la opción predeterminada realizada por el navegador. Algunas propiedades se aplican a todos los elementos y otras se aplican solo a elementos específicos. Asimismo, existen valores adecuados para ser utilizados en cada propiedad.

Para cambiar el color de nuestro párrafo simple, usamos la propiedad `color`. Esta propiedad se refiere al color *foreground*, es decir, de las letras del párrafo. El color en sí va en la parte de valor de la regla y se puede especificar en muchos formatos diferentes, incluidos nombres simples como `red`, `green`, `blue`, `yellow`, etc. Por lo tanto, para hacer la letra de el párrafo `púrpura`, agregue la propiedad personalizada `color: purple` al atributo `style` del elemento:

```
<p style="color: purple">My first stylized paragraph</p>
```

Otras propiedades personalizadas pueden ir en la misma propiedad de estilo, pero deben estar separadas por punto y coma. Si desea aumentar el tamaño de la fuente, por ejemplo, agregue `font-size: larger` a la propiedad `style`:

```
<p style="color: purple; font-size: larger">My first stylized paragraph</p>
```

| | |
|---|---|
| **NOTE** | No es necesario agregar espacios alrededor de los dos puntos y el punto y coma, pero pueden facilitar la lectura del código CSS. |

Para ver el resultado de estos cambios, guarde el siguiente HTML en un archivo y luego ábralo en un navegador web (los resultados se muestran en Figure 32):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
</head>
<body>

<p style="color: purple; font-size: larger">My first stylized paragraph</p>

<p style="color: green; font-size: larger">My second stylized paragraph</p>

</body>
</html>
```

My first stylized paragraph

My second stylized paragraph

*Figure 32. Un cambio visual muy simple usando CSS.*

Imagine cada elemento de la página como un rectángulo o una caja cuyas propiedades geométricas y decoraciones puede manipular con CSS. El mecanismo de renderizado utiliza dos conceptos estándares básicos para la colocación de elementos: colocación *block* y colocación *inline*. Los elementos en bloque ocupan todo el espacio horizontal de su elemento padre y se colocan secuencialmente, de arriba a abajo. Su altura (su *dimensión vertical*, que no debe confundirse con su posición *top* en la página) generalmente depende de la cantidad de contenido que tengan. Los elementos en línea siguen el método de izquierda a derecha similar a los lenguajes escritos occidentales: los elementos se colocan horizontalmente, de izquierda a derecha, hasta que no hay más espacio en el lado derecho, después de lo cual el elemento continúa en una nueva línea justo debajo de la actual. Los elementos como `p`, `div` y `section` se colocan en bloque de forma predeterminada, mientras que los elementos como `span`, `em`, `a` y letras individuales se colocan en línea. Estos métodos básicos de ubicación pueden modificarse fundamentalmente mediante reglas CSS.

El rectángulo correspondiente al elemento `p` en el cuerpo del documento HTML de muestra será visible si agrega la propiedad `background-color` a la regla (Figure 33):

```
<p style="color: purple; font-size: larger; background-color: silver">My first
stylized paragraph</p>

<p style="color: green; font-size: larger; background-color: silver">My second
stylized paragraph</p>
```

My first stylized paragraph

My second stylized paragraph

*Figure 33. Rectángulos correspondientes a los párrafos.*

A medida que agrega nuevas propiedades personalizadas de CSS al atributo `style`, notará que el código comienza a desordenarse. Escribir demasiadas reglas CSS en el atributo `style` socava la separación de estructura (HTML) y presentación (CSS). Además, pronto se dará cuenta de que muchos estilos se comparten entre diferentes elementos y no es aconsejable repetirlos en todos.

### Reglas CSS

En lugar de diseñar los elementos directamente en sus atributos de `style`, es mucho más práctico informar al navegador sobre la colección completa de elementos a los que se aplica el estilo personalizado. Lo hacemos agregando un *selector* a las propiedades personalizadas, haciendo coincidir los elementos por tipo, clase, ID único, posición relativa, etc. La combinación de un *selector* y las propiedades personalizadas correspondientes — también conocidas como *declaraciones* — constituyen reglas *CSS*. La sintaxis básica de una regla CSS es `selector { property: value }`. Como en el atributo `style`, las propiedades separadas por punto y coma pueden agruparse, como en `p { color: purple; font-size: larger }`. Esta regla coincide con todos los elementos `p` de la página y aplica las propiedades personalizadas de `color` y `font-size`.

Una regla CSS para un elemento principal coincidirá automáticamente con todos sus elementos secundarios. Esto significa que podríamos aplicar las propiedades personalizadas a todo el texto de la página, independientemente de si está dentro o fuera de una etiqueta <p>, utilizando el selector `body` en su lugar: `body { color: purple; font-size: larger }`. Esta estrategia nos libera de escribir la misma regla nuevamente para todos sus hijos, pero puede ser necesario escribir reglas adicionales para "deshacer" o modificar las reglas heredadas.

### La etiqueta `style`

La etiqueta <style> nos permite escribir reglas CSS dentro de la página HTML que queremos diseñar. Permite al navegador diferenciar el código CSS del código HTML. La etiqueta <style> va en

la sección `head` del documento:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <style type="text/css">

  p { color: purple; font-size: larger }

  </style>

</head>
<body>

<p>My first stylized paragraph</p>

<p>My second stylized paragraph</p>

</body>
</html>
```

El atributo `type` le especifica al navegador qué tipo de contenido hay dentro de la etiqueta `<style>`, es decir, su *MIME type*. Ya que todos los navegadores que soportan CSS asumen que el tipo de etiqueta `<style>` es `text/css` por defecto, incluyendo el atributo `type` es opcional. También hay un atributo `media` que indica los medios (pantallas de computadora o impresión, por ejemplo) a los que se aplican las reglas CSS en la etiqueta `<style>`. De forma predeterminada, las reglas de CSS se aplican a cualquier medio en el que se presente el documento.

Al igual que en el código HTML, los saltos de línea y la sangría del código no cambian la forma en que el navegador interpreta las reglas CSS. El siguiente fragmento de código:

```
<style type="text/css">

p { color: purple; font-size: larger }

</style>
```

tiene exactamente el mismo resultado que este:

```
<style type="text/css">

p {
  color: purple;
  font-size: larger;
}

</style>
```

Los bytes adicionales utilizados por los espacios y los saltos de línea hacen poca diferencia en el tamaño final del documento y no tienen un impacto significativo en el tiempo de carga de la página, por lo que la elección del diseño es una cuestión de gustos. Tenga en cuenta el punto y coma después de la última declaración (`font-size: larger;`) de la regla CSS. Ese punto y coma no es obligatorio, pero tenerlo allí hace que sea más fácil agregar otra declaración en la siguiente línea sin preocuparse por los puntos y coma que faltan.

Tener las declaraciones en líneas separadas también ayuda cuando necesita comentar una declaración. Siempre que desee deshabilitar temporalmente una declaración por motivos de resolución de problemas, por ejemplo, puede encerrar la línea correspondiente con `/*` y `*/`:

```
p {
  color: purple;
  /*
  font-size: larger;
  */
}
```

o:

```
p {
  color: purple;
  /* font-size: larger; */
}
```

Escrito así, el navegador ignorará la declaración `font-size: larger`. Tenga cuidado de abrir y cerrar los comentarios correctamente, de lo contrario, es posible que el navegador no pueda interpretar las reglas.

Los comentarios también son útiles para escribir recordatorios sobre las reglas:

```
/* Texts inside <p> are purple and larger */
p {
  color: purple;
  font-size: larger;
}
```

Los recordatorios como el del ejemplo son prescindibles en hojas de estilos que contienen una pequeña cantidad de reglas, pero son esenciales para ayudar a navegar por hojas de estilos con cientos (o más) reglas.

Aunque el atributo `style` y la etiqueta `<style>` son adecuados para probar estilos personalizados y útiles para situaciones específicas, no se usan comúnmente. En cambio, las reglas de CSS generalmente se guardan en un archivo separado al que se puede hacer referencia desde el documento HTML.

### CSS en archivos externos

El método preferido para asociar las definiciones CSS con un documento HTML es almacenar el código CSS en un archivo separado. Este método ofrece dos ventajas principales sobre los anteriores:

- Las mismas reglas de estilo se pueden compartir entre distintos documentos.
- El navegador suele almacenar en caché el archivo CSS, lo que mejora los tiempos de carga futuros.

Los archivos CSS tienen la extensión `.css` y, al igual que los archivos HTML, pueden ser editados por cualquier editor de texto plano. A diferencia de los archivos HTML, los archivos CSS no tienen una estructura de nivel superior construida con etiquetas jerárquicas como `<head>` o `<body>`. Más bien, el archivo CSS es solo una lista de reglas procesadas en orden secuencial por el navegador. Las mismas reglas escritas dentro de una etiqueta `<style>` podrían ir en un archivo CSS.

La asociación entre el documento HTML y las reglas CSS almacenadas en un archivo se define solo en el documento HTML. Para el archivo CSS, no importa si existen elementos que coincidan con sus reglas, por lo que no es necesario especificar los documentos HTML a los que está vinculado. En el lado HTML, todas las hojas de estilos vinculadas se aplicarán al documento, como si las reglas estuvieran escritas en una etiqueta `<style>`.

La etiqueta HTML `<link>` define una hoja de estilo externa que se usará en el documento actual y debe ir en la sección `head` del documento HTML:

```
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <link href="style.css" rel="stylesheet">

</head>
```

Ahora puede colocar la regla para el elemento `p` que usamos antes en el archivo `style.css`, y los resultados que verá el visitante de la página web serán los mismos. Si el archivo CSS no está en el mismo directorio que el documento HTML, especifique su ruta relativa o completa en el atributo `href`. La etiqueta `<link>` puede referirse a diferentes tipos de recursos externos, por lo que es importante establecer qué relación tiene el recurso externo con el documento actual. Para archivos CSS externos, la relación se define en `rel="stylesheet"`.

El atributo `media` se puede usar de la misma manera que para la etiqueta `<style>`: indica los medios, como pantallas de computadora o impresión, a los que se deben aplicar las reglas del archivo externo.

CSS puede cambiar completamente un elemento, sin embargo es importante usar el elemento apropiado para los componentes de la página. De lo contrario, es posible que las tecnologías de accesibilidad, como los lectores de pantalla, no puedan identificar las secciones correctas de la página.

## Ejercicios guiados

1. ¿Qué métodos se pueden utilizar para cambiar la apariencia de los elementos HTML mediante CSS?

2. ¿Por qué no se recomienda utilizar el atributo `style` de la etiqueta `<p>` si hay párrafos hermanos que deberían verse iguales?

3. ¿Cuál es la política de ubicación predeterminada para colocar un elemento `div`?

4. ¿Qué atributo de la etiqueta `<link>` indica la ubicación de un archivo CSS externo?

5. ¿Cuál es la sección correcta para insertar el elemento `link` dentro de un documento HTML?

## Ejercicios de exploración

1. ¿Por qué no se recomienda usar una etiqueta `<div>` para identificar una palabra en una oración común?

2. ¿Qué sucede si comienza un comentario con `/*` en medio de un archivo CSS, pero se olvida de cerrarlo con `*/`?

3. Escriba una regla CSS para dibujar un subrayado en todos los elementos `em` del documento.

4. ¿Cómo puede indicar que una hoja de estilos de una etiqueta `<style>` o `<link>` debe ser utilizada solo por sintetizadores de voz?

## Resumen

Esta lección cubre los conceptos básicos de CSS y cómo integrarlos con HTML. Las reglas escritas en hojas de estilos CSS son el método estándar para cambiar la apariencia de los documentos HTML. CSS nos permite mantener el contenido semántico separado de las políticas de presentación. Esta lección abarca los siguientes conceptos y procedimientos:

- Cómo cambiar las propiedades de CSS usando el atributo `style`.
- La sintaxis básica de las reglas CSS.
- Usar la etiqueta `<style>` para incrustar reglas CSS en el documento.
- Usar la etiqueta `<link>` para incorporar hojas de estilo externas al documento.

## Respuestas a los ejercicios guiados

1. ¿Qué métodos se pueden utilizar para cambiar la apariencia de los elementos HTML mediante CSS?

   Tres métodos básicos: Insertar las reglas directamente en la etiqueta del elemento, en una sección separada del código fuente de la página o en un archivo externo para ser referenciado por la página HTML.

2. ¿Por qué no se recomienda utilizar el atributo `style` de la etiqueta `<p>` si hay párrafos hermanos que deberían verse iguales?

   La declaración CSS deberá replicarse en las otras etiquetas `<p>`, lo que requiere mucho tiempo, aumenta el tamaño del archivo y es propenso a errores.

3. ¿Cuál es la política de ubicación predeterminada para colocar un elemento `div`?

   El elemento `div` se trata como un elemento en bloque por defecto, por lo que ocupará todo el espacio horizontal de su elemento padre y su altura dependerá de su contenido.

4. ¿Qué atributo de la etiqueta `<link>` indica la ubicación de un archivo CSS externo?

   El atributo `href`.

5. ¿Cuál es la sección correcta para insertar el elemento `link` dentro de un documento HTML?

   El elemento `link` debe estar en la sección `head` del documento HTML.

## Respuestas a los ejercicios de exploración

1. ¿Por qué no se recomienda usar una etiqueta `<div>` para identificar una palabra en una oración común?

   La etiqueta `<div>` proporciona una separación semántica entre dos secciones distintas del documento e interfiere con las herramientas de accesibilidad cuando se usa para elementos de texto en línea.

2. ¿Qué sucede si comienza un comentario con `/*` en medio de un archivo CSS, pero se olvida de cerrarlo con `*/`?

   Todas las reglas posteriores al comentario serán ignoradas por el navegador.

3. Escriba una regla CSS para dibujar un subrayado en todos los elementos `em` del documento.

   ```
   em { text-decoration: underline }
   ```

   o

   ```
   em { text-decoration-line: underline }
   ```

4. ¿Cómo puede indicar que una hoja de estilos de una etiqueta `<style>` o `<link>` debe ser utilizada solo por sintetizadores de voz?

   El valor de su atributo `media` debe ser `speech`.




# 033.2 Selectores CSS y aplicación de estilo

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 033.2

**Peso**

3

**Áreas de conocimiento clave**

- Use selectores para aplicar reglas CSS a elementos
- Comprender las pseudoclases de CSS
- Comprender el orden de las reglas y la precedencia en CSS
- Comprender la herencia en CSS

**Lista parcial de archivos, términos y utilidades**

- `element;` `.class;` `#id;`
- `a, b;` `a.class;` `a[attr]` `a b;`
- `:hover`, `focus`
- `!important`




# 033.2 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 033 Estilo de contenido CSS |
| **Objetivo:** | 033.2 Selectores CSS y aplicación de estilo |
| **Lección:** | 1 de 1 |

## Introducción

Al escribir una regla CSS, debemos especificarle al navegador a qué elementos se aplica la regla. Lo hacemos especificando un patrón *selector*: que puede coincidir con un elemento o grupo de elementos. Los selectores vienen en muchas formas diferentes, que pueden basarse en el nombre del elemento, sus atributos, su ubicación relativa en la estructura del documento o una combinación de estas características.

### Estilos de toda la página

Una de las ventajas de usar CSS es que no es necesario escribir reglas individuales para elementos que comparten el mismo estilo. Un asterisco aplica la regla a todos los elementos de la página web, como se muestra en el siguiente ejemplo:

```
* {
  color: purple;
  font-size: large;
}
```

El selector `*` no es el único método para aplicar una regla de estilo a todos los elementos de la página. Un selector que simplemente coincide con un elemento por su nombre de etiqueta se denomina *selector de tipo*, por lo que cualquier nombre de etiqueta HTML como `body`, `p`, `table`, `em`, etc., se puede utilizar como selector. En CSS, el estilo del padre es heredado por sus elementos secundarios. Entonces, en la práctica, usar el selector `*` tiene el mismo efecto que aplicar una regla al elemento `body`:

```
body {
  color: purple;
  font-size: large;
}
```

Además, la función en cascada de CSS le permite ajustar las propiedades heredadas de un elemento. Puede escribir una regla CSS general que se aplique a todos los elementos de la página y luego escribir reglas para elementos o conjuntos de elementos específicos.

Si el mismo elemento coincide con dos o más reglas en conflicto, el navegador aplica la regla del selector más específico. Tome las siguientes reglas CSS como ejemplo:

```
body {
  color: purple;
  font-size: large;
}

li {
  font-size: small;
}
```

El navegador aplicará los estilos `color: purple` y `font-size: large` a todos los elementos dentro del elemento `body`. Sin embargo, si hay elementos `li` en la página, el navegador reemplazará el estilo `font-size: large` por el estilo `font-size: small`, porque el selector `li` tiene una relación más fuerte con el `li` que el selector `body`.

CSS no limita el número de selectores equivalentes en la misma hoja de estilos, por lo que puede tener dos o más reglas usando el mismo selector:

```
li {
  font-size: small;
}

li {
  font-size: x-small;
}
```

En este caso, ambas reglas son igualmente específicas para el elemento `li`. El navegador no puede aplicar reglas en conflicto, por lo que elegirá la regla que viene más adelante en el archivo CSS. Para evitar confusiones, la recomendación es agrupar todas las propiedades que utilizan el mismo selector.

El orden en el que aparecen las reglas en la hoja de estilos afecta la forma en que se aplican en el documento, pero puede anular este comportamiento utilizando la regla `!important`. Si, por alguna razón, desea mantener las dos reglas `li` separadas, pero forzar la aplicación de la primera en lugar de la segunda, marque la primera regla como importante:

```
li {
  font-size: small !important;
}

li {
  font-size: x-small;
}
```

Las reglas marcadas con `!important` deben usarse con precaución, ya que rompen la cascada natural de la hoja de estilos y dificultan la búsqueda y corrección de problemas dentro del archivo CSS.

## Selectores restrictivos

Vimos que podemos cambiar ciertas propiedades heredadas usando selectores que coincidan con etiquetas específicas. Sin embargo, normalmente necesitamos usar estilos distintos para elementos del mismo tipo.

Los atributos de las etiquetas HTML se pueden incorporar en los selectores para restringir el conjunto de elementos a los que se refieren. Suponga que la página HTML en la que está trabajando tiene dos tipos de listas desordenadas (`<ul>`): una se usa en la parte superior de la página como un menú a las secciones del sitio web y el otro tipo se usa para listas convencionales en el cuerpo del texto:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

<div id="content">

<p>The three rocky planets of the solar system are:</p>

<ul>
  <li>Mercury</li>
  <li>Venus</li>
  <li>Earth</li>
  <li>Mars</li>
</ul>

<p>The outer giant planets made most of gas are:</p>

<ul>
  <li>Jupiter</li>
  <li>Saturn</li>
  <li>Uranus</li>
  <li>Neptune</li>
</ul>

</div><!-- #content -->


<div id="footer">

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

```
</div><!-- #footer -->

</body>
</html>
```

De forma predeterminada, cada elemento de la lista tiene un círculo negro a su izquierda. Es posible que desee eliminar los círculos de la lista del menú superior y dejar los círculos en la otra lista. Sin embargo, no puede simplemente usar el selector `li` porque hacerlo también eliminará los círculos en la lista dentro de la sección del cuerpo del texto. Necesitará una forma de decirle al navegador que modifique solo los elementos de la lista utilizados en una lista, pero no en la otra.

Hay varias formas de escribir selectores que coincidan con elementos específicos de la página. Como se mencionó anteriormente, primero veremos cómo usar los atributos de los elementos para hacerlo. Para este ejemplo en particular, podemos usar el atributo `id` para especificar solo la lista superior.

El atributo `id` asigna un identificador único al elemento correspondiente, que podemos usar como parte del selector de la regla CSS. Antes de escribir la regla CSS, edite el archivo HTML de muestra y agregue `id="topmenu"` al elemento `ul` utilizado para el menú superior:

```
<ul id="topmenu">
  <li>Home</li>
  <li>Articles</li>
  <li>About</li>
</ul>
```

Ya existe un elemento `link` en la sección `head` del documento HTML que apunta al archivo de hoja de estilos `style.css` en la misma carpeta, por lo que podemos agregarle las siguientes reglas CSS:

```
ul#topmenu {
  list-style-type: none
}
```

El caracter hash se utiliza en un selector, después de un elemento, para designar un ID (sin espacios que los separen). El nombre de la etiqueta a la izquierda del hash es opcional, ya que no habrá ningún otro elemento con el mismo ID. Por lo tanto, el selector podría escribirse como `#topmenu`.

Aunque la propiedad `list-style-type` no es una propiedad directa del elemento `ul`, las propiedades CSS del elemento padre son heredadas por sus hijos, por lo que el estilo asignado al elemento `ul` será heredado por su elementos secundarios `li`.

No todos los elementos tienen un ID mediante el cual se pueden seleccionar. De hecho, se espera que solo algunos elementos clave de diseño de una página tengan ID. Tome las listas de planetas utilizadas en el código de muestra, por ejemplo. Aunque es posible asignar ID únicos para cada elemento repetido individual como estos, este método no es práctico para páginas más extensas con mucho contenido. Más bien, podemos usar el ID del elemento principal `div` como selector para cambiar las propiedades de sus elementos internos.

Sin embargo, el elemento `div` no está directamente relacionado con las listas HTML, por lo que agregarle la propiedad `list-style-type` no tendrá ningún efecto en sus hijos. Por lo tanto, para cambiar el círculo negro en las listas dentro del contenido `div` a un círculo hueco, debemos usar un selector *descendiente*:

```
#topmenu {
  list-style-type: none
}

#content ul {
  list-style-type: circle
}
```

El selector `#content ul` se denomina selector descendiente porque solo coincide con los elementos `ul` que son hijos del elemento cuyo ID es `content`. Podemos utilizar tantos niveles de descendencia como sea necesario. Por ejemplo, el uso de `#content ul li` coincidiría solo con los elementos `li` que son descendientes de los elementos `ul` que son descendientes del elemento cuyo ID es `content`. Pero en este ejemplo, el selector más largo tendrá el mismo efecto que usar `#content ul`, porque los elementos `li` heredan las propiedades CSS establecidas en su padre `ul`. Los selectores descendientes son una técnica esencial a medida que aumenta la complejidad del diseño de la página.

Digamos que ahora desea cambiar la propiedad `font-style` de los elementos de la lista `topmenu` y en la lista del *footer div* para que parezcan oblicuos. No puede simplemente escribir una regla CSS usando `ul` como selector, porque también cambiará los elementos de la lista en el *content div*. Hasta ahora, hemos cambiado las propiedades de CSS usando un selector a la vez, y este método también se puede usar para esta tarea:

```
#topmenu {
  font-style: oblique
}

#footer ul {
  font-style: oblique
}
```

Sin embargo, los selectores separados no son la única forma de hacerlo. CSS nos permite agrupar selectores que comparten uno o más estilos, usando una lista de selectores separados por comas:

```
#topmenu, #footer ul {
  font-style: oblique
}
```

La agrupación de selectores elimina el trabajo adicional de escribir estilos duplicados. Además, es posible que desee cambiar la propiedad nuevamente en el futuro y es posible que no recuerde cambiarla en todos los lugares.

### Clases

Si no desea preocuparse demasiado por la jerarquía de elementos, simplemente puede agregar una `clase` al conjunto de elementos que desea personalizar. Las clases son similares a los ID, pero en lugar de identificar solo un elemento en la página, pueden identificar un grupo de elementos que comparten las mismas características.

Tome la página HTML de muestra en la que estamos trabajando, por ejemplo. Es poco probable que en las páginas del mundo real encontremos estructuras tan simples, por lo que sería más práctico seleccionar un elemento usando solo clases, o una combinación de clases y descendencia. Para aplicar la propiedad `font-style: oblique` a las listas del menú usando clases, primero necesitamos agregar la propiedad `class` a los elementos en el archivo HTML. Lo haremos primero en el menú superior:

```
<ul id="topmenu" class="menu">
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

Y luego en el menú del pie de página:

```
<div id="footer">

<ul class="menu">
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

</div><!-- #footer -->
```

Con eso, podemos reemplazar el grupo selector `#topmenu, #footer ul` por el selector basado en clases `.menu`:

```
.menu {
  font-style: oblique
}
```

Al igual que con los selectores basados en ID, agregar el nombre de la etiqueta a la izquierda del punto en los selectores basados en clases es opcional. Sin embargo, a diferencia de los ID, se supone que la misma clase se usa en más de un elemento y ni siquiera es necesario que sean del mismo tipo. Por lo tanto, si la clase `menu` se comparte entre diferentes tipos de elementos, usar el selector `ul.menu` solo coincidiría con los elementos `ul` que tienen la clase `menu`. En su lugar, usar `.menu` como selector coincidirá con cualquier elemento que tenga la clase `menu`, independientemente de su tipo.

Además, los elementos se pueden asociar a más de una clase. Podríamos diferenciar entre el menú superior e inferior agregando una clase extra a cada uno de ellos:

```
<ul id="topmenu" class="menu top">
```

Y en el menú del pie de página:

```
<ul class="menu footer">
```

Cuando el atributo `class` tiene más de una clase, deben estar separadas por espacios. Ahora, además de la regla CSS compartida entre los elementos de la clase `menu`, podemos seleccionar el menú superior y pie de página usando sus clases correspondientes:

```
.menu {
  font-style: oblique
}

.menu.top {
  font-size: large
}

.menu.footer {
  font-size: small
}
```

Tenga en cuenta que escribir `.menu.top` difiere de `.menu .top` (con un espacio entre las palabras). El primer selector coincidirá con elementos que tengan las clases `menu` y `top`, mientras que el segundo coincidirá con elementos que tengan la clase `top` y un elemento principal con la clase `menu`.

## Selectores especiales

Los selectores CSS también pueden coincidir con los estados dinámicos de los elementos. Estos selectores son particularmente útiles para elementos interactivos, como hipervínculos. Es posible que desee la aparición de hipervínculos cuando el puntero del mouse se mueve sobre ellos, para llamar la atención del visitante.

De regreso a nuestra página de muestra, podríamos eliminar los subrayados de los enlaces en el menú superior y mostrar una línea solo cuando el puntero del mouse se mueva sobre el enlace correspondiente. Para hacer esto, primero escribimos una regla para eliminar el subrayado de los enlaces en el menú superior:

```
.menu.top a {
  text-decoration: none
}
```

Luego usamos la pseudoclase `:hover` en esos mismos elementos para crear una regla CSS que se aplicará solo cuando el puntero del mouse esté sobre el elemento correspondiente:

```
.menu.top a:hover {
  text-decoration: underline
}
```

El selector de pseudoclase `:hover` acepta todas las propiedades de las reglas CSS convencionales.

Otras pseudoclases incluyen `:visited`, que coincide con los hipervínculos que ya se han visitado, y `:focus`, que coincide con los elementos del formulario que han recibido el foco.

# Ejercicios guiados

1. Supongamos que una página HTML tiene una hoja de estilo asociada que contiene las dos reglas siguientes:

   ```
   p {
     color: green;
   }

   p {
     color: red;
   }
   ```

   ¿Qué color aplicará el navegador al texto dentro de los elementos `p`?

2. ¿Cuál es la diferencia entre usar el selector de ID `div#main` y `#main`?

3. ¿Qué selector coincide con todos los elementos `p` usados dentro de un `div` con el atributo de ID `#main`?

4. ¿Cuál es la diferencia entre usar el selector de clases `p.highlight` y `.highlight`?

## Ejercicios de exploración

1. Escriba una sola regla CSS que cambie la propiedad `font-style` a `oblique`. La regla debe coincidir solo con los elementos `a` que están dentro de `<div id="sidebar"></div>` o `<ul class="links"></ul>`.

2. Suponga que desea cambiar el estilo de los elementos cuyo atributo `class` se establece en `article reference`, como en `<p class="article reference">`. Sin embargo, el selector `.article .reference` no parece alterar su apariencia. ¿Por qué el selector no coincide con los elementos como se esperaba?

3. Escriba una regla CSS para cambiar la propiedad `color` de todos los enlaces visitados en la página a `red`.

## Resumen

Esta lección cubre cómo usar selectores CSS y cómo el navegador decide qué estilos aplicar a cada elemento. Al estar separado del marcado HTML, CSS proporciona muchos selectores para hacer coincidir elementos individuales o grupos de elementos en la página. La lección abarca los siguientes conceptos y procedimientos:

- Estilos de página amplia y herencia de estilos.
- Elementos de estilo por tipo.
- Usar el ID del elemento y la clase como selectores.
- Selectores compuestos.
- Uso de pseudoclases para diseñar elementos dinámicamente.

# Respuestas a los ejercicios guiados

1. Supongamos que una página HTML tiene una hoja de estilo asociada que contiene las dos reglas siguientes:

   ```
   p {
     color: green;
   }

   p {
     color: red;
   }
   ```

   ¿Qué color aplicará el navegador al texto dentro de los elementos `p`?

   El color `red`. Cuando dos o más selectores equivalentes tienen propiedades en conflicto, el navegador elegirá el último.

2. ¿Cuál es la diferencia entre usar el selector de ID `div#main` y `#main`?

   El selector `div#main` coincide con un elemento `div` que tiene el ID `main`, mientras que el selector `#main` coincide con el elemento que tiene el ID `main`, independientemente de su tipo.

3. ¿Qué selector coincide con todos los elementos `p` utilizados dentro de un `div` con el atributo ID `#main`?

   El selector `#main p` o `div#main p`.

4. ¿Cuál es la diferencia entre usar el selector de clases `p.highlight` y `.highlight`?

   El selector `p.highlight` solo coincide con los elementos de tipo `p` que tienen la clase `highlight`, mientras que el selector `.highlight` coincide con todos los elementos que tienen la clase `highlight`, independientemente de su tipo.

## Respuestas a los ejercicios de exploración

1. Escribe una única regla CSS que cambie la propiedad `font-style` a `oblique`. La regla debe coincidir solo con los elementos `a` que están dentro de `<div id="sidebar"></div>` o `<ul class="links"></ul>`.

   ```
   #sidebar a, ul.links a {
     font-style: oblique
   }
   ```

2. Suponga que desea cambiar el estilo de los elementos cuyo atributo `class` se establece en `article reference`, como en `<p class="article reference">`. Sin embargo, el selector `.article .reference` no parece alterar su apariencia. ¿Por qué el selector no coincide con los elementos como se esperaba?

   El selector `.article .reference` coincidirá con los elementos que tienen la clase `reference` que son descendientes de elementos que tienen la clase `article`. Para hacer coincidir los elementos que tienen las clases `article` y `reference`, el selector debe ser `.article.reference` (sin el espacio entre ellos).

3. Escriba una regla CSS para cambiar la propiedad `color` de todos los enlaces visitados en la página a `red`

   ```
   a:visited {
     color: red;
   }
   ```




# 033.3 Estilo CSS

### Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 033.3

### Peso

2

### Áreas de conocimiento clave

- Comprender las propiedades fundamentales de CSS
- Comprender las unidades comúnmente utilizadas en CSS

### Lista parcial de archivos, términos y utilidades

- `px`, `%`, `em`, `rem`, `vw`, `vh`
- `color`, `background`, `background-*`, `font`, `font-*`, `text-*`, `list-style`, `line-height`




# 033.3 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 033 Estilo de contenido CSS |
| **Objetivo:** | 033.3 Estilo CSS |
| **Lección:** | 1 de 1 |

## Introducción

CSS proporciona cientos de propiedades que se pueden utilizar para modificar la apariencia de los elementos HTML. Solo los diseñadores experimentados logran recordar la mayoría de ellos. Sin embargo, es práctico conocer las propiedades básicas que son aplicables a cualquier elemento, así como algunas propiedades específicas del elemento. Este capítulo cubre algunas propiedades populares que probablemente utilizará.

### Valores y propiedades comunes de CSS

Muchas propiedades CSS tienen el mismo tipo de valor. Los colores, por ejemplo, tienen el mismo formato numérico, ya sea que cambie el color de fuente o el color de fondo. Del mismo modo, las unidades disponibles para cambiar el tamaño de la fuente también se utilizan para cambiar el grosor de un borde. Sin embargo, el formato del valor no siempre es único. Los colores, por ejemplo, se pueden introducir en varios formatos diferentes, como veremos a continuación.

### Colores

Cambiar el color de un elemento es probablemente una de las primeras cosas que los diseñadores

aprenden a hacer con CSS. Puede cambiar el color del texto, el color de fondo, el color del borde de los elementos y más.

El valor de un color en CSS se puede escribir como una *palabra clave de color* (es decir, un nombre de color) o como un valor numérico que representa cada componente de color. Todos los nombres de colores comunes, como `black`, `white`, `red`, `purple`, `green`, `yellow` y `blue`, se aceptan como palabras claves de colores. La lista completa de palabras clave de color aceptadas por CSS está disponible en la página web de la W3C. Pero para tener un control más preciso sobre el color, puede utilizar el valor numérico.

### Palabras claves de colores

Primero usaremos la palabra clave color, porque es más simple. La propiedad `color` cambia el color del texto en el elemento correspondiente. Entonces, para poner todo el texto de la página en violeta, puede escribir la siguiente regla CSS:

```
* {
  color: purple;
}
```

### Valores numéricos de colores

Aunque son intuitivas, las palabras claves de colores no ofrecen la gama completa de colores posibles en las pantallas modernas. Los diseñadores web suelen desarrollar una paleta que emplea colores personalizados, asignando valores específicos a los componentes rojo, verde y azul.

Cada componente de color es un número binario de ocho bits, que va de 0 a 255. Los tres componentes deben especificarse en la mezcla de colores y su orden es siempre rojo, verde y azul. Por lo tanto, para cambiar el color de todo el texto de la página a rojo usando notación RGB, use `rgb(255,0,0)`:

```
* {
  color: rgb(255,0,0);
}
```

Un componente establecido en `0` significa que el color básico correspondiente no se utiliza en la mezcla de colores. También se pueden utilizar porcentajes en lugar de números:

```
* {
  color: rgb(100%,0%,0%);
}
```

La notación RGB rara vez se ve si usa una aplicación de dibujo para crear diseños o simplemente para elegir sus colores. Más bien, es más común ver colores en CSS expresados como valores *hexadecimales*. Los componentes de color en notación hexadecimal también van de 0 a 255, pero de una manera más sucinta, comenzando con un símbolo de almohadilla `#` y usando una longitud fija de dos dígitos para todos los componentes. El valor mínimo `0` es `00` y el valor máximo `255` es `FF`, por lo que el color `rojo` es `#FF0000`.

**TIP** Si los dígitos de cada componente de un color hexadecimal se repiten, se puede omitir el segundo dígito. El color `rojo`, por ejemplo, se puede escribir con un solo dígito para cada componente: `#F00`.

La siguiente lista muestra la notación RGB y hexadecimal para algunos colores básicos:

| Palabra clave de color | Notación RGB | Valor hexadecimal |
|---|---|---|
| `black` | `rgb(0,0,0)` | `#000000` |
| `white` | `rgb(255,255,255)` | `#FFFFFF` |
| `red` | `rgb(255,0,0)` | `#FF0000` |
| `purple` | `rgb(128,0,128)` | `#800080` |
| `green` | `rgb(0,128,0)` | `#008000` |
| `yellow` | `rgb(255,255,0` | `#FFFF00` |
| `blue` | `rgb(0,0,255)` | `#0000FF` |

Las palabras claves de colores y las letras en valores de color hexadecimales no distinguen entre mayúsculas y minúsculas.

### Opacidad de color

Además de los colores opacos, es posible rellenar los elementos de la página con colores semitransparentes. La opacidad de un color se puede establecer utilizando un cuarto componente en el valor del color. A diferencia de los otros componentes de color, donde los valores son números enteros que van de 0 a 255, la opacidad es una fracción que va de `0` a `1`.

El valor más bajo, `0`, da como resultado un color completamente transparente, lo que lo hace indistinguible de cualquier otro color completamente transparente. El valor más alto, `1`, da como

resultado un color completamente opaco, que es el mismo que el color original sin ninguna transparencia.

Cuando utilice la notación RGB, especifique colores con un componente de opacidad mediante el prefijo `rgba` en lugar de `rgb`. Por lo tanto, para hacer que el color rojo sea semitransparente, use `rgba (255,0,0,0.5)`. El caracter `a` significa *alpha channel*, un término comúnmente utilizado para especificar el componente de opacidad en la jerga de los gráficos digitales.

La opacidad también se puede establecer en notación hexadecimal. Aquí, al igual que los otros componentes de color, la opacidad varía de `00` a `FF`. Por lo tanto, para hacer que el color `rojo` sea semitransparente usando notación hexadecimal, use `#FF000080`.

## Fondo

El color de fondo de los elementos individuales o de toda la página se establece con la propiedad `background-color`. Toma los mismos valores que la propiedad `color`, ya sea como palabras claves o usando la notación RGB/hexadecimal.

Sin embargo, el fondo de los elementos HTML no se limita a los colores. Con la propiedad `background-image` es posible utilizar una imagen como fondo. Los formatos de imagen aceptados son todos los convencionales aceptados por el navegador, como JPEG y PNG.

La ruta a la imagen debe especificarse usando un designador `url()`. Si la imagen que desea usar está en la misma carpeta que el archivo HTML, es suficiente con usar solo su nombre de archivo:

```
body {
  background-image: url("background.jpg");
}
```

En este ejemplo, el archivo de imagen `background.jpg` se utilizará como imagen de fondo para todo el cuerpo de la página. De forma predeterminada, la imagen de fondo se repite si su tamaño no cubre toda la página, comenzando desde la esquina superior izquierda del área correspondiente al elemento que deseamos aplicar las reglas. Este comportamiento se puede modificar con la propiedad `background-repeat`. Si desea que la imagen de fondo se coloque en el área del elemento sin repetirla, use el valor `no-repeat`:

```
body {
  background-image: url("background.jpg");
  background-repeat: no-repeat;
}
```

También puede hacer que la imagen se repita solo en la dirección horizontal (`background-repeat: repeat-x`) o solo en la dirección vertical (`background-repeat: repeat-y`).

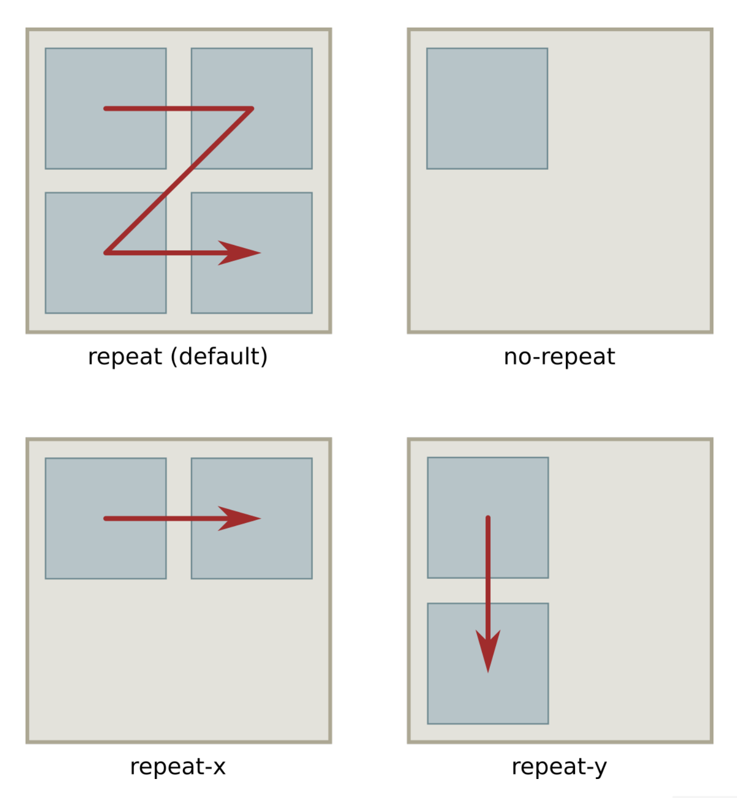



*Figure 34. Colocación de fondo usando la propiedad* `background-repeat`.

> **TIP**
>
> Se pueden combinar dos o más propiedades CSS en una sola propiedad, denominada propiedad background *shorthand*. Las propiedades `background-image` y `background-repeat`, por ejemplo, se pueden combinar en una única propiedad de fondo con `background: no-repeat url("background.jpg")`.

Una imagen de fondo también se puede colocar en una posición específica en el área del elemento usando la propiedad `background-position`. Las cinco posiciones básicas son `top`, `bottom`, `left`, `right` y `center`, pero la posición superior izquierda de la imagen también se puede ajustar con porcentajes:

```
body {
  background-image: url("background.jpg");
  background-repeat: no-repeat;
  background-position: 30% 10%;
}
```

El porcentaje de cada posición es relativo al tamaño correspondiente del elemento. En el ejemplo, el lado izquierdo de la imagen de fondo estará al 30% del ancho del cuerpo (generalmente el cuerpo es

todo el documento visible) y el lado superior de la imagen estará al 10% de la altura del cuerpo.

## Bordes

Cambiar el borde de un elemento también es una personalización de diseño común realizada con CSS. El borde se refiere a la línea que forma un rectángulo alrededor del elemento y tiene tres propiedades básicas: `color`, `style` y `width`.

El color del borde, definido con `border-color`, sigue el mismo formato que vimos para las otras propiedades de color.

Los bordes se pueden trazar con un estilo que no sea una línea continua. Por ejemplo, podría utilizar guiones para el borde con la propiedad `border-style: dashed`. Los otros valores de estilo posibles son:

**`dotted`**
: Una secuencia de puntos redondeados

**`double`**
: Dos líneas rectas

**`groove`**
: Una línea con apariencia tallada

**`ridge`**
: Una línea con apariencia extruida

**`inset`**
: Un elemento que parece incrustado

**`outset`**
: Un elemento que aparece en relieve

La propiedad `border-width` establece el grosor del borde. Su valor puede ser una palabra clave (`thin`, `medium` o `thick`) o un valor numérico específico. Si prefiere utilizar un valor numérico, también deberá especificar su unidad correspondiente. Esto se describe a continuación.

## Valores unitarios

Los tamaños y distancias en CSS se pueden definir de varias formas. Las unidades absolutas se basan en un número fijo de píxeles de la pantalla, por lo que no son tan diferentes de los tamaños y

dimensiones fijos utilizados en los medios impresos. También hay unidades relativas, que se calculan dinámicamente a partir de alguna medida dada por el medio donde se representa la página, como el espacio disponible u otro tamaño escrito en unidades absolutas.

### Unidades absolutas

Las unidades absolutas son equivalentes a sus contrapartes físicas, como centímetros o pulgadas. En las pantallas de ordenador convencionales, una pulgada tendrá 96 píxeles de ancho. Las siguientes unidades absolutas se utilizan comúnmente:

**`in` (inch)**

1 pulgada equivale a 2,54 cm o 96 px.

**`cm` (centimeter)**

1 cm equivale a 96 px / 2,54.

**`mm` (millimeter)**

1 mm equivale a 1 cm / 10.

**`px` (pixel)**

1 px equivale a 1/96 de pulgada.

**`pt` (point)**

1 pt equivale a 1/72 de pulgada.

Tenga en cuenta que la proporción de píxeles por pulgada puede variar. En pantallas de alta resolución, donde los píxeles están más densamente empaquetados, una pulgada corresponderá a más de 96 píxeles.

## Unidades relativas

Las unidades relativas varían según otras medidas o las dimensiones de la ventana gráfica. La ventana gráfica es el área del documento actualmente visible en su ventana. En el modo de pantalla completa, la ventana gráfica corresponde a la pantalla del dispositivo. Las siguientes unidades relativas se utilizan comúnmente:

**`%`**

Porcentaje: es relativo al elemento principal.

`em`
: El tamaño de la fuente utilizada en el elemento.

`rem`
: El tamaño de la fuente utilizada en el elemento raíz.

`vw`
: 1% del ancho de la ventana gráfica.

`vh`
: 1% de la altura de la ventana gráfica.

La ventaja de usar unidades relativas es que puede crear diseños que se pueden ajustar cambiando solo algunos tamaños de clave. Por ejemplo, puede usar la unidad `pt` para establecer el tamaño de fuente en el elemento del cuerpo y la unidad `rem` para las fuentes en otros elementos. Una vez que cambie el tamaño de fuente para el cuerpo, todos los demás tamaños de fuente se ajustarán en consecuencia. Además, el uso de `vw` y `vh` para establecer las dimensiones de las secciones de la página las hace ajustables a pantallas con diferentes tamaños.

## Propiedades de fuentes y texto

La tipografía, o el estudio de los tipos de fuentes, es un tema de diseño muy amplio, y la tipografía CSS no se queda atrás. Sin embargo, existen algunas propiedades de fuente básicas que satisfarán las necesidades de la mayoría de los usuarios que aprenden CSS.

La propiedad `font-family` establece el nombre de la fuente que se utilizará. No hay garantía de que la fuente elegida esté disponible en el sistema donde se verá la página, por lo que esta propiedad puede no tener ningún efecto en el documento. Aunque es posible hacer que el navegador descargue y use el archivo de fuente especificado, la mayoría de los diseñadores web están felices de usar una familia de fuentes genérica en sus documentos.

Las tres familias de fuentes genéricas más comunes son `serif`, `sans-serif` y `monospace`. Serif es la familia de fuentes predeterminada en la mayoría de los navegadores. Si prefiere usar `sans-serif` para toda la página, agregue la siguiente regla a su hoja de estilos:

```
* {
  font-family: sans-serif;
}
```

Opcionalmente, primero puede establecer un nombre de familia de fuentes específico, seguido del

nombre de familia genérico:

```
* {
  font-family: "DejaVu Sans", sans-serif;
}
```

Si el dispositivo que representa la página tiene esa familia de fuentes específica, el navegador la utilizará. De lo contrario, utilizará su fuente predeterminada que coincida con el apellido genérico. Los navegadores buscan fuentes en el orden en que se especifican en la propiedad.

Cualquiera que haya utilizado una aplicación de procesamiento de texto también estará familiarizado con otros tres ajustes de fuente: tamaño, peso y estilo. Estos tres ajustes tienen contrapartes en las propiedades de CSS: `font-size`, `font-weight` y `font-style`.

La propiedad `font-size` acepta tamaños de palabras claves como `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `xxx -grande`. Estas palabras claves son relativas al tamaño de fuente predeterminado que utiliza el navegador. Las palabras claves `larger` y `smaller` cambian el tamaño de la fuente en relación con el tamaño de la fuente del elemento principal. También puede expresar el tamaño de fuente con valores numéricos, incluida la unidad después del valor, o con porcentajes.

Si no desea cambiar el tamaño de la fuente, sino la distancia entre líneas, use la propiedad `line-height`. Un `line-height` de `1` hará que la altura de la línea sea del mismo tamaño que la fuente del elemento, lo que puede hacer que las líneas de texto estén demasiado juntas. Por tanto, un valor superior a 1 es más apropiado para textos. Al igual que la propiedad `font-size`, se pueden usar otras unidades junto con el valor numérico.

El `font-weight` establece el grosor de la fuente con las conocidas palabras claves `normal` o `bold`. Las palabras claves `lighter` y `bold` cambian el peso de la fuente del elemento en relación con el peso de la fuente de su elemento principal.

La propiedad `font-style` se puede establecer en `italic` para seleccionar la versión en cursiva de la familia de fuentes actual. El valor `oblique` selecciona la versión oblicua de la fuente. Estas dos opciones son casi idénticas, pero la versión en cursiva de una fuente suele ser un poco más estrecha que la versión oblicua. Si no existen versiones de la fuente en cursiva ni oblicua, el navegador la inclinará artificialmente.

Hay otras propiedades que cambian la forma en que se representa el texto en el documento. Puede, por ejemplo, agregar un subrayado a algunas partes del texto que desee enfatizar. Primero, use una etiqueta `<span>` para delimitar el texto:

```
<p>CSS is the <span class="under">proper mechanism</span> to style HTML
documents.</p>
```

Luego puede usar el selector `.under` para cambiar la propiedad `text-decoration`:

```
.under {
  text-decoration: underline;
}
```

De forma predeterminada, todos los elementos `a` (enlace) están subrayados. Si desea eliminarlo, use el valor `none` para el `text-decoration` de todos los elementos `a`:

```
a {
  text-decoration: none;
}
```

Al revisar el contenido, a algunos autores les gusta tachar partes incorrectas o desactualizadas del texto, para que el lector sepa que el texto se ha actualizado y lo que se ha eliminado. Para hacerlo, use el valor `line-through` de la propiedad `text-decoration`:

```
.disregard {
  text-decoration: line-through;
}
```

Nuevamente, se puede usar una etiqueta `<span>` para aplicar el estilo:

```
<p>Netscape Navigator <span class="disregard">is</span> was one of the most popular
Web browsers.</p>
```

Otras decoraciones pueden ser específicas de un elemento. Los círculos en las listas de viñetas se pueden personalizar usando la propiedad `list-style-type`. Para cambiarlos a cuadrados, por ejemplo, use `list-style-type: square`. Para simplemente eliminarlos, establezca el valor de `list-style-type` en `none`.

## Ejercicios guiados

1. ¿Cómo podría agregar semitransparencia al color azul (notación RGB `rgb (0,0,255)`) para usarlo en la propiedad CSS `color`?

2. ¿Qué color corresponde al valor hexadecimal `#000`?

3. Dado que `Times New Roman` es una fuente `serif` y que no estará disponible en todos los dispositivos, ¿cómo podría escribir una regla CSS para solicitar `Times New Roman` mientras configura la familia de fuentes genéricas `serif` como alternativa?

4. ¿Cómo podría utilizar una palabra clave de tamaño relativo para establecer el tamaño de fuente del elemento `<p class="disclaimer">` más pequeño en relación con su elemento principal?

## Ejercicios de exploración

1. La propiedad `background` es una abreviatura para establecer más de una propiedad `background-*` a la vez. Reescriba la siguiente regla CSS como una única propiedad abreviada `background`.

```
body {
  background-image: url("background.jpg");
  background-repeat: repeat-x;
}
```

2. Escriba una regla CSS para el elemento `<div id="header"></div>` que cambie *solamente* el ancho del borde inferior a `4px`.

3. Escriba una propiedad `font-size` que duplique el tamaño de fuente utilizado en el elemento raíz de la página.

4. *Double-spacing* es una característica común de formato de texto en los procesadores de texto. ¿Cómo podrías establecer un formato similar usando CSS?

## Resumen

Esta lección cubre la aplicación de estilos simples a elementos de un documento HTML. CSS proporciona cientos de propiedades y la mayoría de los diseñadores web necesitarán recurrir a manuales de referencia para recordarlas todas. No obstante, la mayor parte del tiempo se utiliza un conjunto relativamente pequeño de propiedades y valores, y es importante conocerlos de memoria. La lección abarca los siguientes conceptos y procedimientos:

- Propiedades CSS fundamentales relacionadas con colores, fondos y fuentes.
- Las unidades absolutas y relativas que CSS puede usar para establecer tamaños y distancias, como `px`, `em`, `rem`, `vw`, `vh`, etc.

## Respuestas a los ejercicios guiados

1. ¿Cómo podría agregar semitransparencia al color azul (notación RGB `rgb (0,0,255)`) para usarlo en la propiedad CSS `color`?

   Utilice el prefijo `rgba` y agregue `0.5` como cuarto valor: `rgba (0,0,0,0,5)`.

2. ¿Qué color corresponde al valor hexadecimal `#000`?

   El color `negro`. El valor hexadecimal `#000` es una abreviatura de `#000000`.

3. Dado que `Times New Roman` es una fuente `serif` y que no estará disponible en todos los dispositivos, ¿cómo podría escribir una regla CSS para solicitar `Times New Roman` mientras configura la familia de fuentes genéricas `serif` como reserva?

   ```
   * {
     font-family: "Times New Roman", serif;
   }
   ```

4. ¿Cómo podría usar una palabra clave de tamaño relativo para establecer el tamaño de fuente del elemento `<p class="disclaimer">` más pequeño en relación con su elemento principal?

   Usando la palabra clave `smaller`:

   ```
   p.disclaimer {
     font-size: smaller;
   }
   ```

## Respuestas a los ejercicios de exploración

1. La propiedad `background` es una forma abreviada de establecer más de una propiedad `background-*` a la vez. Reescriba la siguiente regla CSS como una propiedad abreviada única de `background`.

```
body {
  background-image: url("background.jpg");
  background-repeat: repeat-x;
}
```

```
body {
  background: repeat-x url("background.jpg");
}
```

2. Escriba una regla CSS para el elemento `<div id="header"></div>` que cambie *solamente* el ancho del borde inferior a `4px`.

```
#header {
  border-bottom-width: 4px;
}
```

3. Escriba una propiedad `font-size` que duplique el tamaño de fuente utilizado en el elemento raíz de la página.

   La unidad `rem` corresponde al tamaño de fuente utilizado en el elemento raíz, por lo que la propiedad debe ser `font-size: 2rem`.

4. *Double-spacing* es una característica común de formato de texto en los procesadores de texto. ¿Cómo podrías establecer un formato similar usando CSS?

   Establezca la propiedad `line-height` en el valor `2em`, porque la unidad `em` corresponde al tamaño de fuente del elemento actual.




# 033.4 Modelo y diseño CSS

### Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 033.4

### Peso

2

### Áreas de conocimiento clave

- Defina la dimensión, la posición y la alineación de los elementos en un diseño CSS
- Especificar cómo fluye el texto alrededor de otros elementos
- Comprender el flujo de documentos
- Conocimiento del concepto grid CSS
- Conocimiento del diseño web responsivo
- Conocimiento de las consultas de medios CSS

### Lista parcial de archivos, términos y utilidades

- `width`, `height`, `padding`, `padding-*`, `margin`, `margin-*`, `border`, `border-*`
- `top`, `left`, `right`, `bottom`
- `display: block | inline | flex | inline-flex | none`
- `position: static | relative | absolute | fixed | sticky`
- `float: left | right | none`
- `clear: left | right | both | none`




# 033.4 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 033 Estilo de contenido CSS |
| **Objetivo:** | 033.4 Modelo y diseño CSS |
| **Lección:** | 1 de 1 |

## Introducción

Cada elemento visible en un documento HTML se representa como un cuadro rectangular. Por lo tanto, el término *box model* describe el enfoque que adopta CSS para modificar las propiedades visuales de los elementos. Al igual que las cajas de diferentes tamaños, los elementos HTML se pueden anidar dentro de los elementos *container*, generalmente el elemento `div`, para que puedan separarse en secciones.

Podemos utilizar CSS para modificar la posición de los cuadros, desde pequeños ajustes hasta cambios drásticos en la disposición de los elementos en la página. Además del flujo normal, la posición de cada cuadro puede basarse en los elementos que lo rodean, ya sea su relación con su contenedor principal o su relación con el *viewport*, que es el área de la página visible para el usuario. Ningún mecanismo cumple todos los requisitos de diseño posibles, por lo que es posible que necesite una combinación de ellos.

### Flujo Normal

La forma predeterminada en que el navegador representa el árbol del documento se denomina *normal flow (flujo normal)*. Los rectángulos correspondientes a los elementos se colocan más o menos

en el mismo orden en que aparecen en el árbol del documento, en relación con sus elementos principales. No obstante, dependiendo del tipo de elemento, la casilla correspondiente puede seguir distintas reglas de posicionamiento.

Una buena forma de entender la lógica del flujo normal es hacer visibles las cajas. Podemos comenzar con una página muy básica, con solo tres elementos `div` separados, cada uno con un párrafo con texto aleatorio:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CSS Box Model and Layout</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
  <span>id</span> <span>ante</span> <span>tempus</span>
  <span>porta</span> <span>pulvinar</span> <span>et</span>
  <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
  <span>neque.</span> <span>Etiam</span> <span>maximus</span>
  <span>vulputate</span> <span>neque</span> <span>eu</span>
  <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
  <span>felis</span> <span>eget</span> <span>eleifend</span>
  <span>aliquam,</span> <span>dui</span> <span>dolor</span>
  <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultricies</span>
  <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
  <span>pretium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->

</body>
</html>
```

Cada palabra está en un elemento `span`, por lo que podemos diseñar las palabras y ver cómo se tratan también como cuadros. Para que los cuadros sean visibles, debemos editar el archivo de hoja de estilos `style.css` al que hace referencia el documento HTML. Las siguientes reglas harán lo que necesitemos:

```
* {
  font-family: sans;
  font-size: 14pt;
}

div {
  border: 2px solid #00000044;
}

#first {
  background-color: #c4a000ff;
}

#second {
  background-color: #4e9a06ff;
}

#third {
  background-color: #5c3566da;
}

h2 {
  background-color: #ffffff66;
}

p {
  background-color: #ffffff66;
}

span {
  background-color: #ffffffaa;
}
```

El resultado aparece en Figure 35.

**First div**

Sed eget velit id ante tempus porta pulvinar et ex.

**Second div**

Fusce vitae vehicula neque. Etiam maximus vulputate neque eu lobortis. Phasellus condimentum, felis eget eleifend aliquam, dui dolor bibendum leo.

**Third div**

Pellentesque ornare ultricies elementum. Morbi vulputate pretium arcu, sed faucibus.

*Figure 35. El flujo básico de elementos es de arriba a abajo y de izquierda a derecha.*

Figure 35 muestra que cada etiqueta HTML tiene un cuadro correspondiente en el diseño. Los elementos `div`, `h2` y `p` se extienden hasta el ancho de su elemento principal. Por ejemplo, el elemento principal de los elementos `div` es `body`, por lo que se extienden hasta el ancho del cuerpo, mientras que el elemento principal de cada elemento `h2` y `p` es su `div` correspondiente. Los cuadros que se extienden hasta el ancho de su elemento padre se denominan elementos *block*. Algunas de las etiquetas HTML más comunes representadas como bloques son `h1`, `h2`, `h3`, `p`, `ul`, `ol`, `table`, `li`, `div`, `section` , `form` y `aside`. Los elementos en bloque hermanos (elementos en bloque que comparten el mismo elemento padre inmediato) se apilan dentro de su padre de arriba a abajo.

**NOTE** Algunos elementos en bloque no están destinados a ser utilizados como contenedores para otros elementos de bloque. Es posible, por ejemplo, insertar un elemento en bloque dentro de un elemento `h1` o `p`, pero no se considera una buena práctica. Más bien, el diseñador debe usar una etiqueta apropiada como contenedor. Las etiquetas de contenedor comunes son `div`, `section` y `aside`.

Además del texto en sí, elementos como `h1`, `p` y `li` solo esperan elementos *inline* como elementos secundarios. Como la mayoría de las escrituras occidentales, los elementos en línea siguen el flujo de texto de izquierda a derecha. Cuando no queda espacio en el lado derecho, el flujo de elementos en línea continúa en la siguiente línea, como el texto. Algunas etiquetas HTML comunes tratadas como cuadros en línea son `span`, `a`, `em`, `strong`, `img`, `input` y `label`.

En nuestra página HTML de muestra, cada palabra dentro de los párrafos estaba rodeada por una etiqueta `span`, por lo que podían resaltarse con la regla CSS correspondiente. Como se muestra en la imagen, cada elemento `span` se coloca horizontalmente, de izquierda a derecha, hasta que no haya más espacio en el elemento principal.

La altura del elemento depende de su contenido, por lo que el navegador ajusta la altura de un elemento contenedor para acomodar sus elementos de bloque anidados o líneas de elementos en línea. Sin embargo, algunas propiedades de CSS afectan la forma de un cuadro, su posición y la ubicación de sus elementos internos.

Las propiedades `margin` y `padding` afectan a todos los tipos de `box` (cajas). Si no establece estas propiedades explícitamente, el navegador establece algunas de ellas utilizando valores estándares. Como se ve en Figure 23, los elementos `h2` y `p` se renderizaron con un espacio entre ellos. Estos espacios son los márgenes superior e inferior que el navegador agrega a estos elementos de forma predeterminada. Podemos eliminarlos modificando las reglas CSS para los selectores `h2` y `p`:

```
h2 {
  background-color: #ffffff66;
  margin: 0;
}

p {
  background-color: #ffffff66;
  margin: 0;
}
```

El resultado aparece en Figure 36.

**First div**
Sed eget velit id ante tempus porta pulvinar et ex.
**Second div**
Fusce vitae vehicula neque. Etiam maximus vulputate neque eu lobortis. Phasellus condimentum, felis eget eleifend aliquam, dui dolor bibendum leo.
**Third div**
Pellentesque ornare ultricies elementum. Morbi vulputate pretium arcu, sed faucibus.

*Figure 36. La propiedad `margin` puede cambiar o eliminar los márgenes de los elementos.*

El elemento `body` también, por defecto, tiene un pequeño margen que crea un espacio alrededor. Este espacio también se puede eliminar utilizando la propiedad `margin`.

Mientras que la propiedad `margin` define el espacio entre el elemento y su entorno, la propiedad `padding` del elemento define el espacio interno entre los límites del contenedor y sus elementos secundarios. Considere los elementos `h2` y `p` dentro de cada `div` en el código de muestra, por ejemplo. Podríamos usar su propiedad margin para crear un espacio en los bordes del `div` correspondiente, pero es más simple cambiar la propiedad `padding` del contenedor:

```
#second {
  background-color: #4e9a06ff;
  padding: 1em;
}
```

Solo se modificó la regla para el segundo `div`, por lo que los resultados (Figure 37) muestran la diferencia entre el segundo `div` y los otros contenedores `div`.

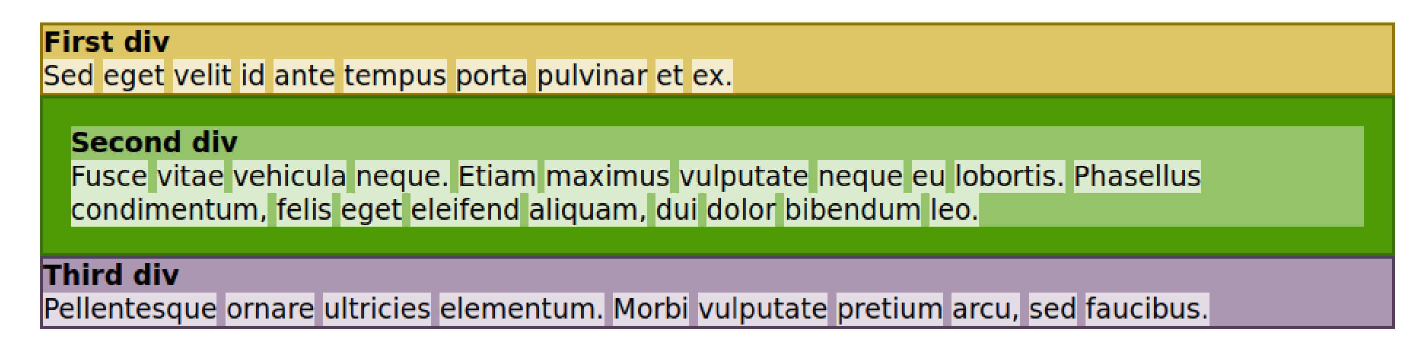



*Figure 37. Los diferentes contenedores `div` pueden tener diferentes `paddings`.*

La propiedad `margin` es una forma abreviada de cuatro propiedades que controlan los cuatro lados del cuadro: `margin-top`, `margin-right`, `margin-bottom` y `margin-left`. Cuando a `margin` se le asigna un valor único, como en los ejemplos hasta ahora, los cuatro márgenes del cuadro lo usan. Cuando se escriben dos valores, el primero define los márgenes superior e inferior, mientras que el segundo define los márgenes derecho e izquierdo. El uso de `margin: 1em 2em`, por ejemplo, define un espacio de 1 em para los márgenes superior e inferior y un espacio de 2 em para los márgenes derecho e izquierdo. Escribir cuatro valores establece los márgenes de los cuatro lados en el sentido de las agujas del reloj, comenzando por la parte superior. No es necesario que los diferentes valores de la propiedad abreviada utilicen las mismas unidades.

La propiedad `padding` también es una forma abreviada, siguiendo los mismos principios que la propiedad `margin`.

En su comportamiento predeterminado, los elementos en bloque se estiran para ajustarse al ancho disponible. Pero esto no es obligatorio. La propiedad `width` puede establecer un tamaño horizontal fijo:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
}
```

La adición de `width: 6em` a la regla CSS encoge el primer `div` horizontalmente, dejando un espacio en blanco en su lado derecho (Figure 38).

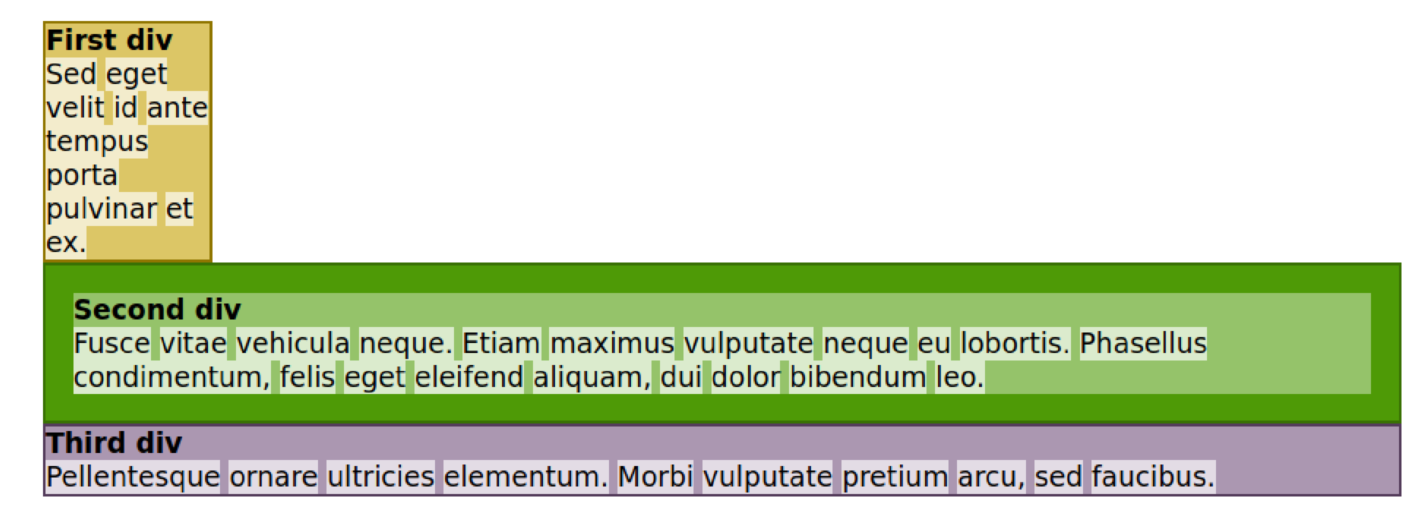



*Figure 38. La propiedad `width` cambia el ancho horizontal del primer `div`.*

En lugar de dejar el primer `div` alineado a la izquierda, es posible que queramos centrarlo. Centrar un cuadro es equivalente a establecer márgenes del mismo tamaño en ambos lados, por lo que podemos usar la propiedad margin para centrarlo. El tamaño del espacio disponible puede variar, por lo que usamos el valor `auto` en los márgenes izquierdo y derecho:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  margin: 0 auto;
}
```

Los márgenes izquierdo y derecho son calculados automáticamente por el navegador y el cuadro se centrará (Figure 39).

**First div**
Sed eget velit id ante tempus porta pulvinar et ex.

**Second div**
Fusce vitae vehicula neque. Etiam maximus vulputate neque eu lobortis. Phasellus condimentum, felis eget eleifend aliquam, dui dolor bibendum leo.

**Third div**
Pellentesque ornare ultricies elementum. Morbi vulputate pretium arcu, sed faucibus.

*Figure 39. La propiedad `margin` se usa para centrar el primer `div`.*

Como se muestra, hacer un elemento en bloque más estrecho no hace que el espacio restante esté disponible para el siguiente elemento. El flujo natural aún se conserva, como si el elemento más estrecho aún ocupara todo el ancho disponible.

## Personalización del flujo normal

El flujo normal es simple y secuencial. CSS también le permite romper el flujo normal y colocar elementos de formas muy específicas, incluso anulando el desplazamiento de la página si lo desea. Veremos varias formas de controlar la posición de los elementos en esta sección.

### Elementos flotantes

Es posible hacer que los elementos de bloques hermanos compartan el mismo espacio horizontal. Una forma de hacerlo es a través de la propiedad `float`, que elimina el elemento del flujo normal. Como sugiere su nombre, la propiedad `float` hace que la caja flote sobre los elementos del bloque que vienen después, por lo que se renderizarán como si estuvieran debajo de la caja flotante. Para hacer que el primer `div` flote a la derecha, agregue `float: right` a la regla CSS correspondiente:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  float: right;
}
```

Los márgenes automáticos se ignoran en un cuadro flotante, por lo que la propiedad `margin` se puede eliminar. Figure 40 muestra el resultado de cómo el primer `div` a la derecha.

**Second div**
Fusce vitae vehicula neque. Etiam maximus vulputate neque eu lobortis. Phasellus condimentum, felis eget eleifend aliquam, dui dolor bibendum leo.
**First div**
Sed eget velit id ante tempus porta pulvinar et ex.
**Third div**
Pellentesque ornare ultricies elementum. Morbi vulputate pretium arcu, sed faucibus.

*Figure 40. El primer `div` está flotando y no forma parte del flujo normal.*

De forma predeterminada, todos los elementos en bloque que vienen después del elemento flotante irán debajo de él. Por lo tanto, dada la altura suficiente, la caja flotante cubrirá todos los elementos restantes del bloque.

Aunque un elemento flotante va por encima de otros elementos en bloque, el contenido en línea dentro del contenedor del elemento flotante se envuelve alrededor del elemento flotante. La inspiración para esto proviene de los diseños de revistas y periódicos, que a menudo envuelven el texto alrededor de una imagen, por ejemplo.

La imagen anterior muestra cómo el primer `div` cubre el segundo `div` y parte del tercer `div`. Supongamos que queremos que el primer `div` flote sobre el segundo `div`, pero no el tercero. La solución es incluir la propiedad `clear` en la regla CSS correspondiente al tercer `div`:

```
#third {
  background-color: #5c3566da;
  clear: right;
}
```

Establecer la propiedad `clear` en `right` hace que el elemento correspondiente omita cualquier elemento anterior flotando a la derecha, reanudando el flujo normal (Figure 41).

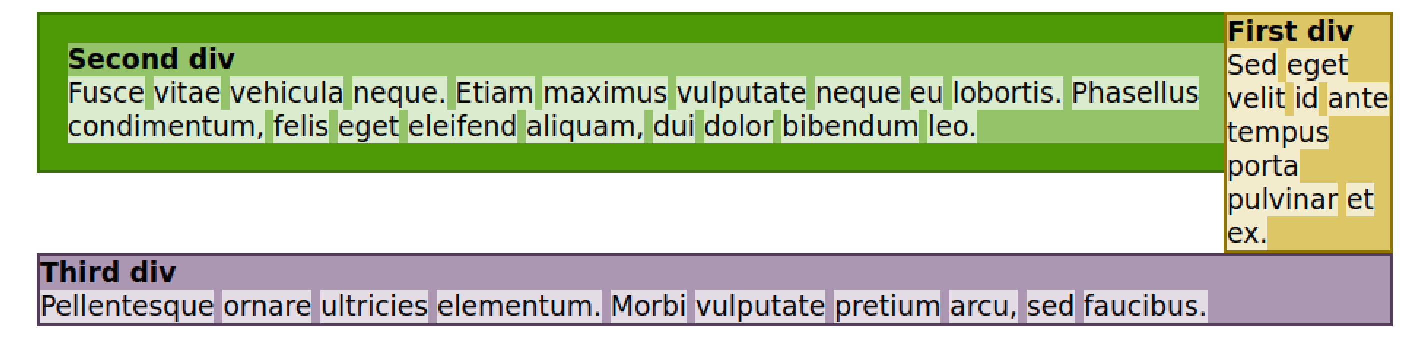



*Figure 41. La propiedad `clear` vuelve al flujo normal.*

Del mismo modo, si un elemento anterior flotaba hacia la izquierda, puede usar `clear: left` para

reanudar el flujo normal. Cuando tenga que omitir elementos flotantes tanto a la izquierda como a la derecha, use `clear: both`.

### Cajas de posicionamiento

En el flujo normal, cada cuadro va después de los cuadros que le preceden en el árbol de documentos. Los elementos hermanos anteriores "empujan" los elementos que vienen después de ellos, moviéndolos hacia la derecha y hacia abajo dentro de su elemento padre. El elemento padre puede tener sus propios hermanos haciéndole lo mismo. Es como colocar azulejos uno al lado del otro en una pared, comenzando por la parte superior.

Este método de colocar las cajas se llama *static*, y es el valor predeterminado para la propiedad CSS `position`. Aparte de definir márgenes y relleno, no hay forma de reposicionar un cuadro estático en la página.

Al igual que los azulejos en la analogía de la pared, la colocación estática no es obligatoria. Al igual que con los mosaicos, puede colocar las cajas en cualquier lugar que desee, incluso cubriendo otras cajas. Para hacerlo, asigne la propiedad `position` uno de los siguientes valores:

**`relative`**
: El elemento sigue el flujo normal del documento, pero puede usar las propiedades `top`, `right`, `bottom` y `left` para establecer desplazamientos relativos a su posición estática original. Las compensaciones también pueden ser negativas. Los otros elementos permanecen en sus lugares originales, como si el elemento relativo aún fuera estático.

**`absolute`**
: El elemento ignora el flujo normal de los otros elementos y se posiciona en la página por las propiedades `top`, `right`, `bottom` e `left`. Sus valores son relativos al cuerpo del documento o a un contenedor principal no estático.

**`fixed`**
: El elemento ignora el flujo normal de los otros elementos y se posiciona por las propiedades `top`, `right`, `bottom` y `left`. Sus valores son relativos a la ventana gráfica (es decir, el área de la pantalla donde se muestra el documento). Los elementos fijos no se mueven cuando el visitante se desplaza por el documento, sino que se asemejan a una pegatina fijada en la pantalla.

**`sticky`**
: El elemento sigue el flujo normal del documento. Sin embargo, en lugar de salir de la ventana gráfica cuando el documento se desplaza, se detendrá en la posición establecida por las propiedades `top`, `right`, `bottom` y `left`. Si el valor de `top` es `10px`, por ejemplo, el elemento dejará de desplazarse por debajo de la parte superior de la ventana gráfica cuando alcance los 10 píxeles

del límite superior de la ventana gráfica. Cuando eso sucede, el resto de la página continúa desplazándose, pero el elemento fijo se comporta como un elemento fijo en esa posición. Volverá a su posición original cuando el documento vuelva a su posición en la ventana gráfica. Los elementos de este tipo se usan comúnmente hoy en día para crear menús superiores que siempre estén visibles.

Las posiciones que pueden usar las propiedades `top`, `right`, `bottom` y `left` no son necesarias para usarlas todas. Si establece las propiedades `top` y `height` de un elemento absoluto, por ejemplo, el navegador calcula implícitamente su propiedad `bottom` (top + height = bottom).

### La propiedad `display`

Si el orden dado por el flujo normal no es un problema en su diseño, pero desea cambiar la forma en que los cuadros se alinean en la página, modifique la propiedad `display` del elemento. La propiedad `display` puede incluso hacer que el elemento desaparezca por completo del documento renderizado, configurando `display: none`. Esto es útil cuando desea mostrar el elemento más tarde usando JavaScript.

La propiedad `display` también puede, por ejemplo, hacer que un elemento de bloque se comporte como un elemento en línea (`display: inline`). Sin embargo, hacerlo no se considera una buena práctica. Existen mejores métodos para colocar elementos contenedores uno al lado del otro, como el *flexbox model*.

El modelo flexbox se inventó para superar las limitaciones del uso de la propiedad `float` y eliminar el uso inadecuado de tablas para estructurar el diseño de la página. Cuando establece la propiedad `display` de un elemento contenedor en `flex` para convertirlo en un contenedor flexbox, sus hijos inmediatos se comportarán más o menos como celdas en una fila de la tabla.

| | |
|---|---|
| **TIP** | Si desea tener aún más control sobre la ubicación de los elementos en la página, eche un vistazo a la función *CSS grid*. La cuadrícula es un poderoso sistema basado en filas y columnas para crear diseños elaborados. |

Para probar flexbox, agregue un nuevo elemento `div` a la página de ejemplo y conviértalo en el contenedor de los tres elementos `div` existentes:

```
<div id="container">

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
  <span>id</span> <span>ante</span> <span>tempus</span>
  <span>porta</span> <span>pulvinar</span> <span>et</span>
  <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
  <span>neque.</span> <span>Etiam</span> <span>maximus</span>
  <span>vulputate</span> <span>neque</span> <span>eu</span>
  <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
  <span>felis</span> <span>eget</span> <span>eleifend</span>
  <span>aliquam,</span> <span>dui</span> <span>dolor</span>
  <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultricies</span>
  <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
  <span>pretium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->


</div><!-- #container -->
```

Agregue la siguiente regla CSS a la hoja de estilo para convertir el contenedor `div` en un contenedor flexbox:

```
#container {
  display: flex;
}
```

El resultado son los tres elementos `div` internos representados uno al lado del otro (Figure 42).

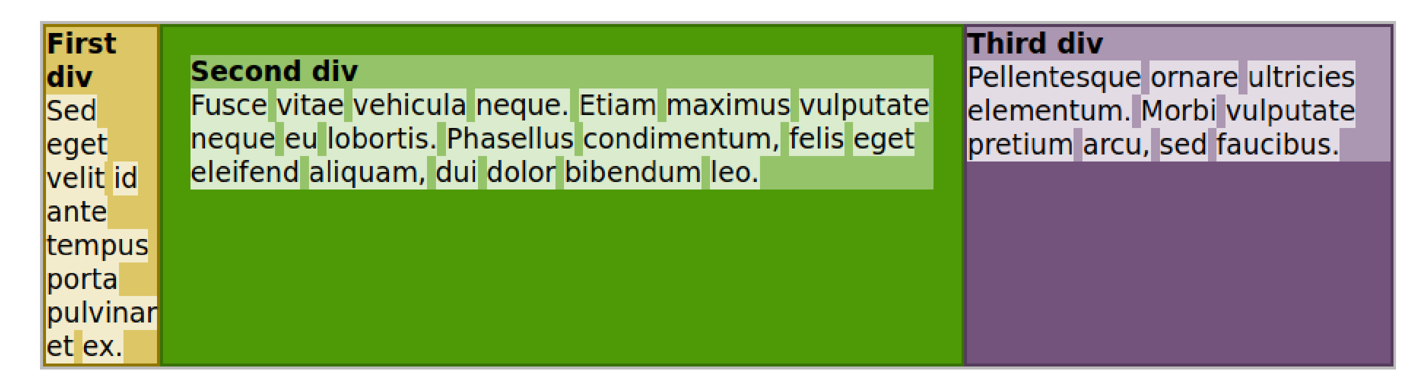



*Figure 42. El modelo flexbox crea una cuadrícula.*

Usar el valor `inline-flex` en lugar de `flex` tiene básicamente el mismo resultado, pero hace que los hijos se comporten más como elementos en línea.

## Diseño Responsivo

Sabemos que CSS proporciona propiedades que ajustan el tamaño de los elementos y las fuentes en relación con el área de pantalla disponible. Sin embargo, es posible que desee ir más allá y utilizar un diseño diferente para diferentes dispositivos: por ejemplo, sistemas de escritorio frente a dispositivos con dimensiones de pantalla inferiores a un determinado tamaño. Este enfoque se llama *diseño web responsivo*, y CSS proporciona métodos llamados *consultas de medios* para hacerlo posible.

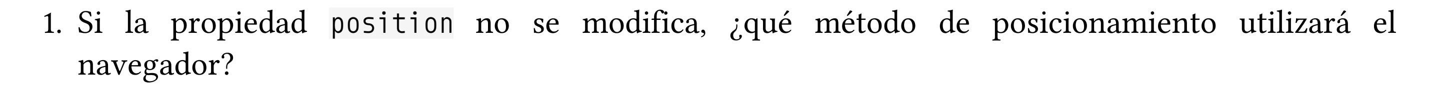
En el ejemplo anterior, modificamos el diseño de la página para colocar los elementos `div` uno al lado del otro en columnas. Ese diseño es adecuado para pantallas más grandes, pero estará demasiado abarrotado en pantallas más pequeñas. Para resolver este problema, podemos agregar una consulta de medios a la hoja de estilos que coincida solo con pantallas con al menos `600px` de ancho:

```
@media (min-width: 600px){
  #container {
    display: flex;
  }
}
```

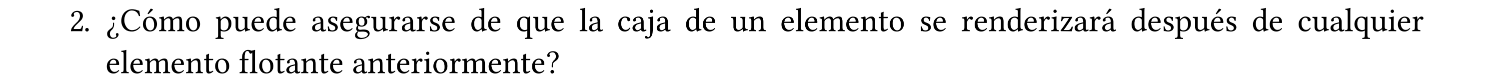
Las reglas CSS dentro de la directiva `@media` se usarán solo si se cumplen los criterios entre paréntesis. En este ejemplo, si el ancho de la ventana gráfica es menor que `600px`, la regla no se aplicará al contenedor `div` y sus elementos secundarios se representarán como elementos `div` convencionales. El navegador reevalúa las consultas de medios cada vez que cambia las dimensiones de la ventana gráfica, por lo que el diseño se puede cambiar en tiempo real mientras se cambia el tamaño de la ventana del navegador o se gira el teléfono inteligente.

## Ejercicios guiados

1. Si la propiedad `position` no se modifica, ¿qué método de posicionamiento utilizará el navegador?



2. ¿Cómo puede asegurarse de que la caja de un elemento se renderizará después de cualquier elemento flotante anteriormente?



3. ¿Cómo se puede utilizar la propiedad abreviada `margin` para establecer los márgenes superior/inferior en `4px` y los márgenes derecho/izquierdo en `6em`?

4. ¿Cómo se puede centrar horizontalmente un elemento contenedor estático con ancho fijo en la página?

## Ejercicios de exploración

1. Escriba una regla CSS que coincida con el elemento `<div class="picture">` para que el texto dentro de sus siguientes elementos de bloque se asiente hacia su lado derecho.

2. ¿Cómo afecta la propiedad `top` a un elemento estático en relación con su elemento padre?

3. ¿Cómo afecta el cambio de la propiedad `display` de un elemento a `flex` su ubicación en el flujo normal?

4. ¿Qué función de CSS le permite utilizar un conjunto de reglas independiente en función de las dimensiones de la pantalla?

## Resumen

Esta lección cubre el modelo de caja CSS y cómo podemos personalizarlo. Además del flujo normal del documento, el diseñador puede hacer uso de diferentes mecanismos de posicionamiento para implementar un diseño personalizado. La lección abarca los siguientes conceptos y procedimientos:

- El flujo normal del documento.
- Ajustes al margen y relleno de la caja de un elemento.
- Uso de las propiedades flotantes y transparentes.
- Mecanismos de posicionamiento: estático, relativo, absoluto, fijo y pegajoso.
- Valores alternativos para la propiedad `display`.
- Conceptos básicos de diseño receptivo.

## Respuestas a los ejercicios guiados

1. Si la propiedad `position` no se modifica, ¿qué método de posicionamiento utilizará el navegador?

   El método `static`.

2. ¿Cómo puede asegurarse de que la caja de un elemento se renderizará después de cualquier elemento flotante anteriormente?

   La propiedad `clear` del elemento debe establecerse en `both`.

3. ¿Cómo se puede utilizar la propiedad abreviada `margin` para establecer los márgenes superior/inferior en `4px` y los márgenes derecho/izquierdo en `6em`?

   Puede ser `margin: 4px 6em` o `margin: 4px 6em 4px 6em`.

4. ¿Cómo se puede centrar horizontalmente un elemento contenedor estático con ancho fijo en la página?

   Usando el valor `auto` en sus propiedades `margin-left` y `margin-right`.

## Respuestas a los ejercicios de exploración

1. Escriba una regla CSS que coincida con el elemento `<div class="picture">` para que el texto dentro de sus siguientes elementos de bloque se asiente hacia su lado derecho.

   `.picture { float: left; }`

2. ¿Cómo afecta la propiedad `top` a un elemento estático en relación con su elemento padre?

   La propiedad `top` no se aplica a los elementos estáticos.

3. ¿Cómo afecta el cambio de la propiedad `display` de un elemento a `flex` su ubicación en el flujo normal?

   La ubicación del elemento en sí no cambia, pero sus elementos secundarios inmediatos se mostrarán uno al lado del otro de forma horizontal.

4. ¿Qué función de CSS le permite utilizar un conjunto de reglas independiente en función de las dimensiones de la pantalla?

   Las *consultas de medios (Media queries)* permiten que el navegador verifique las dimensiones de la ventana gráfica antes de aplicar una regla CSS.




# Tema 034: Programación JavaScript




# 034.1 Ejecución y sintaxis de JavaScript

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 034.1

**Peso**

1

**Áreas de conocimiento clave**

- Comprender el concepto y la estructura del DOM
- Cambiar los contenidos y propiedades de los elementos HTML a través del DOM
- Cambiar el estilo CSS de los elementos HTML a través del DOM
- Activar funciones de JavaScript a partir de elementos HTML

**Lista parcial de archivos, términos y utilidades**

- `<script>`, incluidos los atributos `type` (`text/javascript`) y `src`
- `;`
- `//`, `/* */`
- `console.log`




# 034.1 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 034 Programación JavaScript |
| **Objetivo:** | 034.1 Ejecución y sintaxis de JavaScript |
| **Lección:** | 1 de 1 |

## Introducción

Las páginas web se desarrollan utilizando tres tecnologías estándares: HTML, CSS y JavaScript. JavaScript es un lenguaje de programación que permite al navegador actualizar dinámicamente el contenido del sitio web. El código Javascript generalmente lo ejecuta el mismo navegador que se usa para ver una página web. Esto significa que, al igual que CSS y HTML, el comportamiento exacto de cualquier código que escriba puede diferir entre los navegadores. Pero la mayoría de los navegadores habituales se adhieren a la especificación ECMAScript. Este es un estándar que unifica el uso de JavaScript en la web y será la base de la lección, junto con la especificación HTML5, que determina lo necesario para que un navegador ejecute el Código Javascript en una página web.

### Ejecutar JavaScript en el navegador

Para ejecutar JavaScript, el navegador necesita obtener el código directamente, como parte del HTML que forma la página web, o como una URL que indica una ubicación para que se ejecute un script.

El siguiente ejemplo muestra cómo incluir código directamente en el archivo HTML:

```
<html>
  <head>
  </head>
  <body>
    <h1>Website Headline</h1>
    <p>Content</p>

    <script>
      console.log('test');
    </script>

  </body>
</html>
```

El código se coloca entre las etiquetas `<script>` y `</script>`. Todo lo incluido en estas etiquetas será ejecutado por el navegador directamente al cargar la página.

La posición del elemento `<script>` dentro de la página dicta cuándo se ejecutará. Un documento HTML se analiza de arriba a abajo, y el navegador decide cuándo mostrar los elementos en la pantalla. En el ejemplo que se acaba de mostrar, las etiquetas `<h1>` y `<p>` del sitio web se analizan, y probablemente se muestran, antes de que se ejecute el script. Si el código JavaScript dentro de la etiqueta `<script>` tardara mucho en ejecutarse, la página aún se mostraría sin ningún problema. Sin embargo, si el script se hubiera colocado antes de las otras etiquetas, el visitante de la página web tendría que esperar hasta que el script termine de ejecutarse para ver su contenido. Por esta razón, las etiquetas `<script>` generalmente se colocan en uno de dos lugares:

- Al final del cuerpo HTML, de modo que el script sea lo último que se ejecute. Haga esto cuando el código agregue algo a la página que no sería útil sin el contenido. Un ejemplo sería agregar funcionalidad a un botón, ya que el botón debe existir para que la funcionalidad tenga sentido.
- Dentro del elemento `<head>` del HTML. Esto asegura que la secuencia de comandos se ejecute antes de que se analice el cuerpo HTML. Si desea cambiar el comportamiento de carga de la página, o tiene algo que debe ejecutarse mientras la página aún no está completamente cargada, puede poner el script aquí. Además, si tiene varios scripts que dependen de otro en particular, puede poner ese script compartido en el encabezado para asegurarse de que se ejecute antes que otros.

Por una variedad de razones, incluida la capacidad de administración, es útil colocar el código JavaScript en archivos separados que existen fuera del código HTML. Los archivos JavaScript externos se incluyen mediante una etiqueta `<script>` con un atributo `src`, de la siguiente manera:

```
<html>
  <head>
    <script src="/button-interaction.js"></script>
  </head>
  <body>
  </body>
</html>
```

La etiqueta `src` le especifica al navegador la ubicación de la fuente, es decir, el archivo que contiene el código JavaScript. La ubicación puede ser un archivo en el mismo servidor, como en el ejemplo anterior, o cualquier URL accesible desde la web como `https://www.lpi.org/example.js`. El valor del atributo `src` sigue la misma convención que la importación de archivos CSS o de imágenes, ya que puede ser relativo o absoluto. Al encontrar una etiqueta `script` con el atributo `src`, el navegador intentará obtener el archivo fuente mediante una solicitud HTTP `GET`, por lo que los archivos externos deben ser accesibles.

Cuando utiliza el atributo `src`, cualquier código o texto que se coloque entre las etiquetas `<script>…</script>` se ignora, de acuerdo con la especificación HTML.

```
<html>
  <head>
    <script src="/button-interaction.js">
      console.log("test"); // <-- This is ignored
    </script>
  </head>
  <body>
  </body>
</html>
```

Hay otros atributos que puede agregar a la etiqueta `script` para especificar con más detalle cómo el navegador debe obtener los archivos y cómo debe manejarlos después. La siguiente lista detalla los atributos más importantes:

**`async`**

Puede usarse en etiquetas `script` e indica al navegador que cargue el script en segundo plano, para no bloquear el proceso de carga de la página. La carga de la página seguirá siendo interrumpida después de que el navegador obtenga el script, porque el navegador tiene que analizarlo, lo que se hace inmediatamente una vez que el script se ha recuperado por completo. Este atributo es booleano, por lo que escribir la etiqueta como `<script async src="/script.js"></script>` es suficiente y no es necesario proporcionar ningún valor.

**`defer`**

Similar a `async`, esto le indica al navegador que no bloquee el proceso de carga de la página mientras obtiene el script. En lugar de eso, el navegador aplazará el análisis del script. El navegador esperará hasta que se haya analizado todo el documento HTML y solo entonces analizará el script, antes de anunciar que el documento se ha cargado por completo. Al igual que `async`, `defer` es un atributo booleano y se usa de la misma manera. Dado que `defer` implica `async`, no es útil especificar ambas etiquetas juntas.

> **NOTE** Cuando una página se analiza por completo, el navegador indica que está lista para mostrarse activando un evento `DOMContentLoaded`, cuando el visitante podrá ver el documento. Por lo tanto, el JavaScript incluido en un evento `<head>` siempre tiene la oportunidad de actuar en la página antes de que se muestre, incluso si incluye el atributo `defer`.

**`type`**

Denota qué tipo de script debe esperar el navegador dentro de la etiqueta. El valor predeterminado es JavaScript (`type="application/javascript"`), por lo que este atributo no es necesario cuando se incluye código JavaScript o se apunta a un recurso JavaScript con la etiqueta `src`. Generalmente, se pueden especificar todos los tipos MIME, pero el navegador solo ejecutará los scripts que se denotan como JavaScript. Hay dos casos de uso realistas para este atributo: indicarle al navegador que no ejecute el script configurando `type` en un valor arbitrario como `template` u `other`, o decirle al navegador que el script es un módulo ES6. No cubrimos los módulos ES6 en esta lección.

> **WARNING** Cuando varios scripts tienen el atributo `async`, se ejecutarán en el orden en que terminan de descargarse, *no* en el orden de las etiquetas `script` en el documento. El atributo `defer`, por otro lado, mantiene el orden de las etiquetas `script`.

## Consola del navegador

Aunque normalmente se ejecuta como parte de un sitio web, hay otra forma de ejecutar JavaScript: a través de la consola del navegador. Todos los navegadores de escritorio modernos proporcionan un menú a través del cual puede ejecutar código JavaScript en el motor JavaScript de éstos. Esto generalmente se hace para probar código nuevo o para depurar sitios web existentes.

Hay varias formas de acceder a la consola del navegador, en dependencia de éste. La forma más sencilla es mediante atajos de teclado. Los siguientes son los atajos de teclado para algunos de los navegadores actualmente en uso:

**Chrome**

`Ctrl` + `Shift` + `J` (`Cmd` + `Option` + `J` en Mac)

**Firefox**

`Ctrl` + `Shift` + `K` (`Cmd` + `Option` + `K` en Mac)

**Safari**

`Ctrl` + `Shift` + `?` (`Cmd` + `Option` + `?` en Mac)

También puede hacer clic derecho en una página web y seleccionar la opción "Inspeccionar" o "Inspeccionar elemento" para abrir el inspector, que es otra herramienta del navegador. Cuando se abra el inspector, aparecerá un nuevo panel. En este panel, puede seleccionar la pestaña "Consola", que abrirá la consola del navegador.

Una vez que abra la consola, puede ejecutar JavaScript en la página escribiendo el código directamente en el campo de entrada. El resultado de cualquier código ejecutado se mostrará en una línea separada.

## Declaraciones de JavaScript

Ahora que sabemos cómo comenzar a ejecutar un script, cubriremos los conceptos básicos de cómo se ejecuta realmente un script. Un script JavaScript es una colección de sentencias y bloques. Una declaración de ejemplo es `console.log('test')`. Esta instrucción le indica al navegador que envíe la palabra `test` a la consola.

Cada declaración en JavaScript termina con un punto y coma (`;`). Esto indica al navegador que la declaración está lista y que se puede iniciar una nueva. Considere el siguiente script:

```
var message = "test"; console.log(message);
```

Hemos escrito dos declaraciones. Cada declaración termina con un punto y coma o con el final de la secuencia de comandos. Por motivos de legibilidad, podemos poner las declaraciones en líneas separadas. De esta forma, el script también podría escribirse como:

```
var message = "test";
console.log(message);
```

Esto es posible porque se ignoran todos los espacios en blanco entre declaraciones, como un espacio, una nueva línea o un tab. Los espacios en blanco también se pueden colocar entre palabras claves

individuales dentro de declaraciones, pero esto se explicará con más detalle en una próxima lección. Las declaraciones también pueden estar vacías o compuestas solo de espacios en blanco.

Si una declaración no es válida porque no ha sido terminada con un punto y coma, ECMAScript intenta insertar automáticamente los puntos y comas adecuados, basándose en un conjunto complejo de reglas. La regla más importante es: si una declaración no válida se compone de dos declaraciones válidas separadas por una nueva línea, inserte un punto y coma en la nueva línea. Por ejemplo, el siguiente código no forma una declaración válida:

```
console.log("hello")
console.log("world")
```

Pero un navegador moderno lo ejecutará automáticamente como si estuviera escrito con el punto y coma adecuado:

```
console.log("hello");
console.log("world");
```

Por tanto, es posible omitir el punto y coma en determinados casos. Sin embargo, como las reglas para la inserción automática de punto y coma son complejas, le recomendamos que siempre termine correctamente sus declaraciones para evitar errores no deseados.

## Comentarios de JavaScript

Los scripts grandes pueden volverse bastante complicados. Es posible que desee comentar sobre lo que está escribiendo, para que el script sea más fácil de leer para otras personas o para usted mismo en el futuro. Alternativamente, es posible que desee incluir metainformación en el script, como información de derechos de autor o sobre cuándo se escribió el script y por qué.

Para que sea posible incluir dicha metainformación, JavaScript admite *comentarios*. Un desarrollador puede incluir caracteres especiales en un script que denoten ciertas partes del mismo como un comentario, que se omitirá en la ejecución. La siguiente es una versión muy comentada del script que vimos anteriormente.

```
/*
 This script was written by the author of this lesson in May, 2020.
 It has exactly the same effect as the previous script, but includes comments.
*/

// First, we define a message.
var message = "test";

console.log(message); // Then, we output the message to the console.
```

Los comentarios no son declaraciones y no es necesario terminar con un punto y coma. Hay dos formas de escribir comentarios en JavaScript:

**Multi-line comment**

Use `/*` and `*/` to signal the start and end of a multi-line comment. Everything after `/*`, until the first occurrence of `*/` is ignored. This kind of comment is generally used to span multiple lines, but it can also be used for single lines, or even within a line, like so:

```
console.log(/* what we want to log: */ "hello world")
```

Debido a que el objetivo de los comentarios generalmente es aumentar la legibilidad de un script, debe evitar usar este estilo de comentario dentro de una línea.

**Comentario de una sola línea**

Use `//` (dos barras diagonales) para *comentar* una línea. Todo lo que sigue a la barra doble en la misma línea se ignora. En el ejemplo mostrado anteriormente, este patrón se usa primero para comentar una línea completa. Después de la instrucción `console.log(message);`, se usa para escribir un comentario en el resto de la línea.

En general, los comentarios de una sola línea deben usarse para líneas simples y los comentarios de varias líneas cuando el comentario sea relativamente extenso, incluso si es posible usarlos de otras formas. Deben evitarse los comentarios dentro de una declaración.

Los comentarios también se pueden utilizar para eliminar temporalmente líneas de código real, de la siguiente manera:

```
// We temporarily want to use a different message
// var message = "test";
var message = "something else";
```

## Ejercicios guiados

1. Cree una variable llamada `ColorName` y asígnele el valor `RED`.

| |
|---|
| |

2. ¿Cuáles de los siguientes scripts son válidos?

| | |
|---|---|
| `console.log("hello")`<br>`console.log("world");` | |
| `console.log("hello");`<br>`console.log("world");` | |
| `// console.log("hello")`<br>`console.log("world");` | |
| `console.log("hello");`<br>`console.log("world") //;` | |
| `console.log("hello"); /*`<br>`console.log("world") */` | |

## Ejercicios de exploración

1. ¿Cuántas declaraciones de JavaScript se pueden escribir en una sola línea sin usar un punto y coma?

2. Cree dos variables llamadas `x` y `y`, luego imprima su suma en la consola.

## Resumen

En esta lección, aprendimos diferentes formas de ejecutar JavaScript y cómo modificar el comportamiento de carga del script. También aprendimos los conceptos básicos de composición y comentarios de scripts, y hemos aprendido a usar el comando `console.log()`.

HTML utilizado en esta lección:

**<script>**
: La etiqueta `script` puede usarse para incluir JavaScript directamente o especificando un archivo con el atributo `src`. Modifica cómo se carga el script con los atributos `async` y `defer`.

Conceptos de JavaScript presentados en esta lección:

**`;`**
: Se utiliza un punto y coma para separar declaraciones. El punto y coma a veces se puede omitir, pero no se recomienda.

**`//`, `/*…*/`**
: Los comentarios se pueden utilizar para agregar comentarios o metainformación en un script, o para evitar que se ejecuten declaraciones.

**`console.log("text")`**
: El comando `console.log()` se puede usar para enviar texto a la consola del navegador.

## Respuestas a los ejercicios guiados

1. Cree una variable llamada `ColorName` y asígnele el valor `RED`.

```
var ColorName = "RED";
```

2. ¿Cuáles de los siguientes scripts son válidos?

| | |
|---|---|
| `console.log("hello")`<br>`console.log("world");` | No válido: El primer comando `console.log()` no está terminado correctamente y la línea en su conjunto no forma una declaración válida. |
| `console.log("hello");`<br>`console.log("world");` | Válido: cada declaración se termina correctamente. |
| `// console.log("hello")`<br>`console.log("world");` | Válido: Se ignora todo el código porque es un comentario. |
| `console.log("hello");`<br>`console.log("world") //;` | No válido: a la última declaración le falta un punto y coma. El punto y coma al final se ignora porque está comentado. |
| `console.log("hello"); /*`<br>`console.log("world") */` | Válido: una declaración válida va seguida de un código comentado, que se ignora. |

## Respuestas a los ejercicios de exploración

1. ¿Cuántas declaraciones de JavaScript se pueden escribir en una sola línea sin usar un punto y coma?

   Si estamos al final de una secuencia de comandos, podemos escribir una declaración y se terminará al final del archivo. De lo contrario, no puede escribir una declaración sin un punto y coma con la sintaxis que aprendió hasta ahora.

2. Cree dos variables llamadas `x` e `y`, luego imprima su suma en la consola.

   ```
   var x = 5;
   var y = 10;
   console.log(x+y);
   ```




# 034.2 Estructuras de datos en JavaScript

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 034.2

**Peso**

3

**Áreas de conocimiento clave**

- Definir y usar variables y constantes
- Comprender los tipos de datos
- Comprender la conversión/coerción de tipos
- Comprender matrices y objetos
- Conciencia del alcance de la variable

**Lista parcial de archivos, términos y utilidades**

- `=`, `+`, `-`, `*`, `/`, `%`, `--`, `++`, `+=`, `-=`, `*=`, `/=`
- `var`, `let`, `const`
- `boolean`, `number`, `string`, `symbol`
- `array`, `object`
- `undefined`, `null`, `NaN`




# 034.2 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 034 Programación JavaScript |
| **Objetivo:** | 034.2 Estructuras de datos en JavaScript |
| **Lección:** | 1 de 1 |

## Introducción

Los lenguajes de programación, como los lenguajes naturales, representan la realidad a través de símbolos que se combinan en declaraciones significativas. La realidad representada por un lenguaje de programación son los recursos de la máquina, como las operaciones del procesador, los dispositivos y la memoria.

Cada uno de los innumerables lenguajes de programación adopta un paradigma para representar información. JavaScript adopta las convenciones típicas de los lenguajes de *alto nivel*, donde la mayoría de los detalles, como la asignación de memoria, están implícitos, lo que permite al programador centrarse en el propósito del script en el contexto de la aplicación.

### Lenguajes de alto nivel

Los lenguajes de alto nivel proporcionan reglas abstractas para que el programador necesite escribir menos código para expresar una idea. JavaScript ofrece formas convenientes de hacer uso de la memoria de la computadora, utilizando conceptos de programación que simplifican la escritura de prácticas recurrentes y que generalmente son suficientes para el propósito del desarrollador web.

**NOTE** Aunque es posible utilizar mecanismos especializados para un acceso meticuloso a la memoria, los tipos de datos más simples que veremos son de uso más general.

Las operaciones típicas en una aplicación web consisten en solicitar datos a través de alguna instrucción JavaScript y almacenarlos para ser procesados y eventualmente presentados al usuario. Este almacenamiento es bastante flexible en JavaScript, con formatos de almacenamiento adecuados para cada propósito.

## Declaración de constantes y variables

La declaración de constantes y variables para contener datos es la piedra angular de cualquier lenguaje de programación. JavaScript adopta la convención de la mayoría de los lenguajes de programación, asignando valores a constantes o variables con la sintaxis `name = value`. La constante o variable de la izquierda toma el valor de la derecha. El nombre de la constante o variable debe comenzar con una letra o un guión bajo.

No es necesario indicar el tipo de datos almacenados en la variable, porque JavaScript es un lenguaje de escritura *dinámico*. El tipo de variable se infiere del valor que se le asigna. Sin embargo, es conveniente designar ciertos atributos en la declaración para garantizar el resultado esperado.

**NOTE** TypeScript es un lenguaje inspirado en JavaScript que, al igual que los lenguajes de bajo nivel, le permite declarar variables para tipos específicos de datos.

### Constantes

Una constante es un símbolo que se asigna una vez cuando se inicia el programa y nunca cambia. Las constantes son útiles para especificar valores fijos, como definir la constante `PI` como 3.14159265, o `COMPANY_NAME` para contener el nombre de su empresa.

En una aplicación web, por ejemplo, tome un cliente que recibe información meteorológica de un servidor remoto. El programador puede decidir que la dirección del servidor debe ser constante, porque no cambiará durante la ejecución de la aplicación. Sin embargo, la información de temperatura puede cambiar con cada nueva llegada de datos del servidor.

El intervalo entre consultas realizadas al servidor también se puede definir como una constante, que se puede consultar desde cualquier parte del programa:

```
const update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Cuando se invoca, la función `setup_app()` muestra el mensaje `Update every 10 minutes` en la consola. El término `const` colocado antes del nombre `update_interval` asegura que su valor seguirá siendo el mismo durante toda la ejecución del script. Si se intenta restablecer el valor de una constante, se emite un error `TypeError: Assignment to constant variable`.

### Variables

Sin el término `const`, JavaScript asume automáticamente que `update_interval` es una variable y que su valor puede modificarse. Esto es equivalente a declarar la variable explícitamente con `var`:

```
var update_interval;
update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Tenga en cuenta que aunque la variable `update_interval` se definió fuera de la función, se accedió desde dentro de la función. Cualquier constante o variable declarada fuera de funciones o bloques de código definidos por llaves (`{}`) tiene *alcance global*; es decir, se puede acceder a ellla desde cualquier parte del código. Lo contrario no es cierto: una constante o variable declarada dentro de una función tiene *ámbito local*, por lo que es accesible solo desde el interior de la propia función. Los bloques de código delimitados por corchetes, como los que se encuentran en las estructuras de decisión `if` o los bucles `for`, delimitan el alcance de las constantes, pero no de las variables declaradas como `var`. El siguiente código, por ejemplo, es válido:

```
var success = true;
if ( success == true )
{
  var message = "Transaction succeeded";
  var retry = 0;
}
else
{
  var message = "Transaction failed";
  var retry = 1;
}

console.log(message);
```

La declaración `console.log (message)` puede acceder a la variable `message`, aunque se haya declarado dentro del bloque de código de la estructura `if`. No sucedería lo mismo si `message` fuera constante, como se muestra en el siguiente ejemplo:

```
var success = true;
if ( success == true )
{
  const message = "Transaction succeeded";
  var retry = 0;
}
else
{
  const message = "Transaction failed";
  var retry = 1;
}

console.log(message);
```

En este caso, se emitiría un mensaje de error de tipo `ReferenceError: message is not defined` y se detendría el script. Si bien puede parecer una limitación, restringir el alcance de las variables y constantes ayuda a evitar la confusión entre la información procesada en el cuerpo del script y en sus diferentes bloques de código. Por esta razón, las variables declaradas con `let` en lugar de `var` también tienen un alcance restringido por los bloques delimitados por llaves. Existen otras diferencias sutiles entre declarar una variable con `var` o con `let`, pero la más significativa se refiere al alcance de la variable, como se analiza aquí.

## Tipos de valores

La mayoría de las veces, el programador no necesita preocuparse por los tipos de datos almacenados en variables, porque JavaScript los identifica automáticamente con uno de sus tipos *primitivos* durante la primera asignación de un valor a la variable. Sin embargo, algunas operaciones pueden ser específicas de un tipo de datos u otro y pueden dar lugar a errores cuando se utilizan sin criterio. Además, JavaScript ofrece tipos *estructurados* que le permiten combinar más de un tipo primitivo en un solo objeto.

### Tipos primitivos

Las instancias de tipo primitivo corresponden a variables tradicionales, que almacenan solo un valor. Los tipos se definen implícitamente, por lo que el operador `typeof` se puede utilizar para identificar qué tipo de valor se almacena en una variable:

```
console.log("Undefined variables are of type", typeof variable);

{
  let variable = true;
  console.log("Value `true` is of type " + typeof variable);
}

{
  let variable = 3.14159265;
  console.log("Value `3.14159265` is of type " + typeof variable);
}

{
  let variable = "Text content";
  console.log("Value `Text content` is of type " + typeof variable);
}

{
  let variable = Symbol();
  console.log("A symbol is of type " + typeof variable);
}
```

Este script mostrará en la consola qué tipo de variable se utilizó en cada caso:

```
Variables indefinidas son del tipo undefined
Valor `true` es de tipo booleano
Valor `3.114159265` es del tipo numérico
Valor `Text content` es del tipo string
Un símbolo es del tipo symbol
```

Observe que la primera línea intenta encontrar el tipo de una variable no declarada. Esto hace que la variable dada se identifique como `undefined`. El tipo `symbol` es el primitivo menos intuitivo. Su propósito es proporcionar un nombre de atributo único dentro de un objeto cuando no es necesario definir un nombre de atributo específico. Un objeto es una de las estructuras de datos que veremos a continuación.

### Tipos estructurados

Si bien los tipos primitivos son suficientes para escribir rutinas simples, existen inconvenientes en su uso exclusivo en aplicaciones más complejas. Una aplicación de comercio electrónico, por ejemplo, sería mucho más difícil de escribir, porque el programador necesitaría encontrar formas de almacenar listas de elementos y valores correspondientes utilizando solo variables con tipos primitivos.

Los tipos estructurados simplifican la tarea de agrupar información de la misma naturaleza en una sola variable. Una lista de artículos en un carrito de compras, por ejemplo, se puede almacenar en una sola variable de tipo *array*:

```
let cart = ['Milk', 'Bread', 'Eggs'];
```

Como se demuestra en el ejemplo, una serie de elementos se designa con corchetes. El ejemplo ha llenado el arreglo con tres valores de cadena literales, de ahí el uso de comillas simples. Las variables también se pueden usar como elementos en un arreglo, pero en ese caso deben designarse sin comillas. El número de elementos en un arreglo se puede consultar con la propiedad `length`:

```
let cart = ['Milk', 'Bread', 'Eggs'];
console.log(cart.length);
```

El número `3` se mostrará en la salida de la consola. Se pueden agregar nuevos elementos al arreglo con el método `push ()`:

```
cart.push('Candy');
console.log(cart.length);
```

Esta vez, la cantidad mostrada será `4`. Se puede acceder a cada elemento de la lista por su índice numérico, comenzando con `0`:

```
console.log(cart[0]);
console.log(cart[3]);
```

La salida que se muestra en la consola será:

```
Milk
Candy
```

Así como puede usar `push()` para agregar un elemento, puede usar `pop()` para eliminar el último elemento de un arreglo.

No es necesario que los valores almacenados en un arreglo sean del mismo tipo. Es posible, por ejemplo, almacenar la cantidad de cada artículo a su lado. Una lista de la compra como la del ejemplo anterior se podría construir de la siguiente manera:

```
let cart = ['Milk', 1, 'Bread', 4, 'Eggs', 12, 'Candy', 2];

// Item indexes are even
let item = 2;

// Quantities indexes are odd
let quantity = 3;

console.log("Item: " + cart[item]);
console.log("Quantity: " + cart[quantity]);
```

La salida que se muestra en la consola después de ejecutar este código es:

```
Item: Bread
Quantity: 4
```

Como ya habrá notado, combinar los nombres de los elementos con sus respectivas cantidades en un

solo arreglo puede no ser una buena idea, porque la relación entre ellos no es explícita en la estructura de datos y es muy susceptible a errores (humanos). Incluso si se usara un arreglo para los nombres y otro aarreglo para las cantidades, mantener la integridad de la lista requeriría el mismo cuidado y no sería muy productivo. En estas situaciones, la mejor alternativa es utilizar una estructura de datos más apropiada: un *objeto*.

En JavaScript, una estructura de datos de tipo objeto le permite vincular propiedades a una variable. Además, a diferencia de un arreglo, los elementos que componen un objeto no tienen un orden fijo. Un artículo de la lista de compras, por ejemplo, puede ser un objeto con las propiedades `name` y `quantity`:

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
console.log("Quantity: " + item.quantity);
```

Este ejemplo muestra que un objeto se puede definir usando llaves (`{}`), donde cada par de propiedad/valor está separado por dos puntos y las propiedades por comas. La propiedad es accesible en el formato *variable.property*, como en `item.name`, tanto para lectura como para asignar nuevos valores. La salida que se muestra en la consola después de ejecutar este código es:

```
Item: Milk
Quantity: 1
```

Finalmente, cada objeto que representa un artículo puede incluirse en el arreglo de la lista de compras. Esto se puede hacer directamente al crear la lista:

```
let cart = [{ name: 'Milk', quantity: 1 }, { name: 'Bread', quantity: 4 }];
```

Como antes, un nuevo objeto que representa un elemento se puede agregar más tarde al arreglo:

```
cart.push({ name: 'Eggs', quantity: 12 });
```

Ahora se accede a los elementos de la lista por su índice y su nombre de propiedad:

```
console.log("Third item: " + cart[2].name);
console.log(cart[2].name + " quantity: " + cart[2].quantity);
```

La salida que se muestra en la consola después de ejecutar este código es:

```
third item: eggs
Eggs quantity: 12
```

Las estructuras de datos permiten al programador mantener su código mucho más organizado y fácil de mantener, ya sea por el autor original o por otros programadores del equipo. Además, muchas salidas de las funciones de JavaScript están en tipos estructurados, que deben ser manejados correctamente por el programador.

## Operadores

Hasta ahora, prácticamente solo hemos visto cómo asignar valores a las variables recién creadas. Tan simple como es, cualquier programa realizará varias manipulaciones en los valores de las variables. JavaScript ofrece varios tipos de *operadores* que pueden actuar directamente sobre el valor de una variable o almacenar el resultado de la operación en una nueva variable.

La mayoría de los operadores están orientados a operaciones aritméticas. Para aumentar la cantidad de un artículo en la lista de compras, por ejemplo, simplemente use el operador de adición `+`:

```
item.quantity = item.quantity + 1;
```

El siguiente fragmento imprime el valor de `item.quantity` antes y después de la adición. No mezcle las funciones del signo más en el fragmento. Las declaraciones `console.log` usan un signo más para combinar dos cadenas.

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
console.log("Quantity: " + item.quantity);

item.quantity = item.quantity + 1;
console.log("New quantity: " + item.quantity);
```

La salida que se muestra en la consola después de ejecutar este código es:

```
Item: Milk
Quantity: 1
New quantity: 2
```

Tenga en cuenta que el valor almacenado previamente en `item.quantity` se utiliza como operando de suma: `item.quantity = item.quantity + 1`. Solo después de que se completa la operación, el valor en `item.quantity` se actualiza con el resultado de la operación. Este tipo de operación aritmética que involucra el valor actual de la variable de destino es bastante común, por lo que existen operadores abreviados que le permiten escribir la misma operación en un formato reducido:

```
item.quantity += 1;
```

Las otras operaciones básicas también tienen operadores abreviados equivalentes:

- `a = a - b` es equivalente a `a -= b`.
- `a = a * b` es equivalente a `a *= b`.
- `a = a / b` es equivalente a `a /= b`.

Para la suma y la resta, hay un tercer formato disponible cuando el segundo operando es solo una unidad:

- `a = a + 1` es equivalente a `a++`.
- `a = a - 1` es equivalente a `a--`.

Se puede combinar más de un operador en la misma operación y el resultado se puede almacenar en una nueva variable. Por ejemplo, la siguiente declaración calcula el precio total de un artículo más un costo de envío:

```
let total = item.quantity * 9.99 + 3.15;
```

El orden en el que se realizan las operaciones sigue el orden de precedencia tradicional: primero se realizan las operaciones de multiplicación y división, y solo entonces se realizan las operaciones de suma y resta. Los operadores con la misma precedencia se ejecutan en el orden en que aparecen en la expresión, de izquierda a derecha. Para anular el orden de precedencia predeterminado, puede utilizar paréntesis, como en `a * (b + c)`.

En algunas situaciones, el resultado de una operación ni siquiera necesita almacenarse en una variable. Este es el caso cuando desea evaluar el resultado de una expresión dentro de una declaración `if`:

```
if ( item.quantiy % 2 == 0 )
{
  console.log("Quantity for the item is even");
}
else
{
  console.log("Quantity for the item is odd");
}
```

El operador `%` (módulo) devuelve el resto de la división del primer operando por el segundo operando. En el ejemplo, la instrucción `if` verifica si el resto de la división de `item.quantity` por `2` es igual a cero, es decir, si `item.quantity` es un múltiplo de 2.

Cuando uno de los operandos del operador `+` es una cadena, los otros operandos son *coercidos* en cadenas y el resultado es una concatenación de cadenas. En los ejemplos anteriores, este tipo de operación se utilizó para concatenar cadenas y variables en el argumento de la declaración `console.log`.

Es posible que esta conversión automática no sea el comportamiento deseado. Un valor proporcionado por el usuario en un campo de formulario, por ejemplo, puede identificarse como una cadena, pero en realidad es un valor numérico. En casos como este, la variable debe convertirse primero en un número con la función `Number()`:

```
sum = Number(value1) + value2;
```

Además, es importante verificar que el usuario haya proporcionado un valor válido antes de continuar con la operación. En JavaScript, una variable sin un valor asignado contiene el valor `null`. Esto permite al programador usar una declaración de decisión como `if ( value1 == null )` para verificar si a una variable se le asignó un valor, independientemente del tipo de valor asignado a la variable.

## Ejercicios guiados

1. Un arreglo es una estructura de datos presente en varios lenguajes de programación, algunos de los cuales solo permiten arreglos con elementos del mismo tipo. En el caso de JavaScript, ¿es posible definir un arreglo con elementos de diferentes tipos?

2. Basado en el ejemplo `let item = { name: 'Milk', quantity: 1 }` para un objeto en una lista de compras, ¿cómo podría declararse este objeto para incluir el precio del artículo?

3. En una sola línea de código, ¿cuáles son las formas de actualizar el valor de una variable a la mitad de su valor actual?

## Ejercicios de exploración

1. En el siguiente código, ¿qué valor se mostrará en la salida de la consola?

```
var value = "Global";

{
  value = "Location";
}

console.log(value);
```

2. ¿Qué sucederá cuando uno o más de los operandos involucrados en una operación de multiplicación sea una cadena?

3. ¿Cómo es posible eliminar el elemento `Eggs` de la matriz `cart` declarada con `let cart = ['Milk', 'Bread', 'Eggs']`?

## Resumen

Esta lección cubre el uso básico de constantes y variables en JavaScript. JavaScript es un *lenguaje de escritura dinámico*, por lo que el programador no necesita especificar el tipo de variable antes de configurarlo. Sin embargo, es importante conocer los tipos primitivos del lenguaje para asegurar el resultado correcto de las operaciones básicas. Además, las estructuras de datos como arreglos y objetos combinan tipos primitivos y permiten al programador construir variables compuestas más complejas. Esta lección abarca los siguientes conceptos y procedimientos:

- Comprensión de constantes y variables
- Alcance de variables
- Declaración de variables con `var` y `let`
- Tipos primitivos
- Operadores aritméticos
- Arreglos y objetos
- Coerción y conversión de tipos

## Respuestas a los ejercicios guiados

1. Un arreglo es una estructura de datos presente en varios lenguajes de programación, algunos de los cuales solo permiten arreglos con elementos del mismo tipo. En el caso de JavaScript, ¿es posible definir un arreglo con elementos de diferentes tipos?

   Sí, en JavaScript es posible definir arreglos con elementos de diferentes tipos primitivos, como cadenas y números.

2. Basado en el ejemplo `let item = { name: 'Milk', quantity: 1 }` para un objeto en una lista de compras, ¿cómo podría declararse este objeto para incluir el precio del artículo?

   `let item = { name: 'Milk', quantity: 1, price: 4.99 };`

3. En una sola línea de código, ¿cuáles son las formas de actualizar el valor de una variable a la mitad de su valor actual?

   Se puede usar la propia variable como operando, `value = value / 2`, o el operador abreviado `/=`: `value /= 2`.

## Respuestas a los ejercicios de exploración

1. En el siguiente código, ¿qué valor se mostrará en la salida de la consola?

   ```
   var value = "Global";

   {
     value = "Location";
   }

   console.log(value);
   ```

   `Location`

2. ¿Qué sucederá cuando uno o más de los operandos involucrados en una operación de multiplicación sea una cadena?

   JavaScript asignará el valor `NaN` (No es un número) al resultado, lo que indica que la operación no es válida.

3. ¿Cómo es posible eliminar el elemento `Eggs` de la matriz `cart` declarada con `let cart = ['Milk', 'Bread', 'Eggs']`?

   Las matrices en javascript tienen el método `pop()`, que elimina el último elemento de la lista: `cart.pop()`.




# 034.3 Funciones y estructuras de control de JavaScript

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 034.3

**Peso**

4

**Áreas de conocimiento clave**

- Comprender los valores verdaderos y falsos.
- Comprender los operadores de comparación
- Comprender la diferencia entre comparación flexible y estricta
- Usar condicionales
- Usar bucles
- Definir funciones personalizadas

**Lista parcial de archivos, términos y utilidades**

- `if`, `else if`, `else`
- `switch`, `case`, `break`
- `for`, `while`, `break`, `continue`
- `function`, `return`
- `==`, `!=`, `<`, `▯`, `>`, `>=`
- `===`, `!==`




# 034.3 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 034 Programación JavaScript |
| **Objetivo:** | 034.3 Funciones y estructuras de control de JavaScript |
| **Lección:** | 1 de 2 |

## Introducción

Como cualquier otro lenguaje de programación, el código JavaScript es una colección de declaraciones que le indica al intérprete de instrucciones qué hacer en orden secuencial. Sin embargo, esto no significa que cada instrucción deba ejecutarse solo una vez o que deban ejecutarse en absoluto. La mayoría de las declaraciones deben ejecutarse solo cuando se cumplen condiciones específicas. Incluso cuando una secuencia de comandos se activa de forma asincrónica por eventos independientes, a menudo tiene que comprobar una serie de variables de control para averiguar la parte correcta del código a ejecutar.

### Estructuras If

La estructura de control más simple viene dada por la instrucción `if`, que ejecutará la instrucción inmediatamente después si la condición especificada es verdadera. JavaScript considera que las condiciones son verdaderas si el valor evaluado es distinto de cero. Todo lo que esté entre paréntesis después de la palabra `if` (los espacios se ignoran) se interpretará como una condición. En el siguiente ejemplo, el número literal `1` es la condición:

```
if ( 1 ) console.log("1 is always true");
```

El número `1` está escrito explícitamente en esta condición de ejemplo, por lo que se trata como un valor constante (permanece igual durante la ejecución del script) y siempre dará como resultado verdadero cuando se use como expresión condicional. La palabra `true` también podría usarse en lugar de `1`, ya que el lenguaje también la trata como un valor verdadero literal. La instrucción `console.log` imprime sus argumentos en la *ventana de la consola* del navegador.

**TIP**

La consola del navegador muestra errores, advertencias y mensajes informativos enviados con la instrucción de JavaScript `console.log`. En Chrome, la combinación de teclas `Ctrl` + `Shift` + `J` (`Cmd` + `Option` + `J` en Mac) abre la consola. En Firefox, la combinación de teclas `Ctrl` + `Shift` + `K` (`Cmd` + `Option` + `K` en Mac) abre la pestaña de la consola en las herramientas del desarrollador.

Aunque sintácticamente correcto, el uso de expresiones constantes en condiciones no es muy útil. En una aplicación real, probablemente querrá probar la veracidad de una variable:

```
let my_number = 3;
if ( my_number ) console.log("The value of my_number is", my_number, "and it yields
true");
```

El valor asignado a la variable `my_number` (`3`) es distinto de cero, por lo que arroja verdadero. Pero este ejemplo no es de uso común, porque rara vez es necesario probar si un número es igual a cero. Es mucho más común comparar un valor con otro y probar si el resultado es verdadero:

```
let my_number = 3;
if ( my_number == 3 ) console.log("The value of my_number is", my_number, "indeed
");
```

El operador de comparación doble igual se utiliza porque el operador igual simple ya está definido como operador de asignación. El valor a cada lado del operador se llama *operand*. El orden de los operandos no importa y cualquier expresión que devuelva un valor puede ser un operando. Aquí una lista de otros operadores de comparación disponibles:

**`value1 == value2`**

True if `value1` is equal to `value2`.

**`value1 != value2`**

True if `value1` is not equal to `value2`.

**`value1 < value2`**

True if `value1` is less than `value2`.

**`value1 > value2`**

True if `value1` is greater than `value2`.

**`value1 <= value2`**

True if `value1` is less than or equal to `value2`.

**`value1 >= value2`**

True if `value1` is grater than or equal to `value2`.

Por lo general, no importa si el operando a la izquierda del operador es una cadena de texto y el operando a la derecha es un número, siempre que JavaScript pueda convertir la expresión en una comparación significativa. Por lo tanto, la cadena que contiene el caracter `1` se tratará como el número 1 en comparación con una variable numérica. Para asegurarse de que la expresión sea verdadera solo si ambos operandos son exactamente del mismo tipo y valor, se debe usar el operador de identidad estricto `===` en lugar de `==`. Del mismo modo, el operador estricto sin identidad `!==` se evalúa como verdadero si el primer operando no es exactamente del mismo tipo y valor que el segundo operador.

Opcionalmente, la estructura de control `if` puede ejecutar una declaración alternativa cuando la expresión se evalúa como falsa:

```
let my_number = 4;
if ( my_number == 3 ) console.log("The value of my_number is 3");
else console.log("The value of my_number is not 3");
```

La instrucción `else` debe seguir inmediatamente a la instrucción `if`. Hasta ahora, estamos ejecutando solo una declaración cuando se cumple la condición. Para ejecutar más de una instrucción, debe encerrarlas entre llaves:

```
let my_number = 4;
if ( my_number == 3 )
{
  console.log("The value of my_number is 3");
  console.log("and this is the second statement in the block");
}
else
{
  console.log("The value of my_number is not 3");
  console.log("and this is the second statement in the block");
}
```

Un grupo de una o más declaraciones delimitadas por un par de llaves se conoce como *declaración de bloque*. Es común usar sentencias de bloque incluso cuando solo hay una instrucción para ejecutar, para que el estilo de codificación sea consistente en todo el script. Además, JavaScript no requiere que las llaves o cualquier declaración estén en líneas separadas, pero hacerlo mejora la legibilidad y facilita el mantenimiento del código.

Las estructuras de control se pueden anidar unas dentro de otras, pero es importante no mezclar las llaves de apertura y cierre de cada declaración de bloque:

```
let my_number = 4;

if ( my_number > 0 )
{
  console.log("The value of my_number is positive");

  if ( my_number % 2 == 0 )
  {
    console.log("and it is an even number");
  }
  else
  {
    console.log("and it is an odd number");
  }
} // end of if ( my_number > 0 )
else
{
  console.log("The value of my_number is less than or equal to 0");
  console.log("and I decided to ignore it");
}
```

Las expresiones evaluadas por la instrucción `if` pueden ser más elaboradas que las comparaciones simples. Este es el caso en el ejemplo anterior, donde se empleó la expresión aritmética `my_number % 2` entre paréntesis en el `if` anidado. El operador `%` devuelve el resto después de dividir el número de su izquierda por el número de su derecha. Los operadores aritméticos como `%` tienen prioridad sobre los operadores de comparación como `==`, por lo que la comparación utilizará el resultado de la expresión aritmética como su operando izquierdo.

En muchas situaciones, las estructuras condicionales anidadas se pueden combinar en una sola estructura utilizando *operadores lógicos*. Si solo estuviéramos interesados en números pares positivos, por ejemplo, se podría usar una única estructura `if`:

```
let my_number = 4;

if ( my_number > 0 && my_number % 2 == 0 )
{
  console.log("The value of my_number is positive");
  console.log("and it is an even number");
}
else
{
  console.log("The value of my_number either 0, negative");
  console.log("or it is a negative number");
}
```

El operador doble `&&` en la expresión evaluada es el operador lógico *AND*. Se evalúa como verdadera solo si la expresión a su izquierda y la expresión a su derecha se evalúan como verdadera. Si desea hacer coincidir números que sean positivos o pares, se debe usar el operador `||` en su lugar, que representa el operador lógico *OR*:

```
let my_number = -4;

if ( my_number > 0 || my_number % 2 == 0 )
{
  console.log("The value of my_number is positive");
  console.log("or it is a even negative number");
}
```

En este ejemplo, solo los números impares negativos no cumplirán los criterios impuestos por la expresión compuesta. Si tiene la intención opuesta, es decir, para hacer coincidir solo los números impares negativos, agregue el operador lógico *NOT* `!` al comienzo de la expresión:

```
let my_number = -5;

if ( ! ( my_number > 0 || my_number % 2 == 0 ) )
{
  console.log("The value of my_number is an odd negative number");
}
```

Agregar el paréntesis en la expresión compuesta obliga a que la expresión encerrada entre ellos se evalúe primero. Sin estos paréntesis, el operador NOT se aplicaría solo a `my_number > 0` y luego se evaluaría la expresión OR. Los operadores `&&` y `||` se conocen como operadores lógicos *binarios* porque requieren dos operandos. `!` se conoce como un operador lógico *unario*, porque solo requiere un operando.

## Estructuras Switch

Aunque la estructura `if` es bastante versátil y suficiente para controlar el flujo del programa, la estructura de control "switch" puede ser más apropiada cuando es necesario evaluar resultados que no sean verdaderos o falsos. Por ejemplo, si queremos tomar una acción distinta para cada elemento elegido de una lista, será necesario escribir una estructura `if` para cada evaluación:

```
// Available languages: en (English), es (Spanish), pt (Portuguese)
let language = "pt";

// Variable to register whether the language was found in the list
let found = 0;

if ( language == "en" )
{
  found = 1;
  console.log("English");
}

if ( found == 0 && language == "es" )
{
  found = 1;
  console.log("Spanish");
}

if ( found == 0 && language == "pt" )
{
  found = 1;
  console.log("Portuguese");
}

if ( found == 0 )
{
  console.log(language, " is unknown to me");
}
```

En este ejemplo, una variable auxiliar `found` es utilizada por todas las estructuras `if` para averiguar si se ha producido una coincidencia. En un caso como éste, la estructura `switch` realizará la misma tarea, pero de una manera más breve:

```
switch ( language )
{
  case "en":
    console.log("English");
    break;
  case "es":
    console.log("Spanish");
    break;
  case "pt":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}
```

Cada `case` anidado se llama *cláusula*. Cuando una cláusula coincide con la expresión evaluada, ejecuta las declaraciones que siguen a los dos puntos hasta la declaración `break`. La última cláusula no necesita una declaración `break` y se usa a menudo para establecer la acción predeterminada cuando no ocurren otras coincidencias. Como se ve en el ejemplo, la variable auxiliar no es necesaria en la estructura `switch`.

| | |
|---|---|
| **WARNING** | `switch` usa una comparación estricta para hacer coincidir expresiones con sus cláusulas `case`. |

Si más de una cláusula desencadena la misma acción, puede combinar dos o más condiciones de `case`:

```
switch ( language )
{
  case "en":
  case "en_US":
  case "en_GB":
    console.log("English");
    break;
  case "es":
    console.log("Spanish");
    break;
  case "pt":
  case "pt_BR":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}
```

## Bucles

En ejemplos anteriores, las estructuras `if` y `switch` eran adecuadas para tareas que deben ejecutarse solo una vez después de pasar una o más pruebas condicionales. Sin embargo, hay situaciones en las que una tarea debe ejecutarse repetidamente, en un llamado *bucle*, siempre que su expresión condicional siga siendo verdadera. Si necesita saber si un número es primo, por ejemplo, deberá verificar si dividir ese número por cualquier número entero mayor que 1 y menor que él mismo tiene un resto igual a 0. Si es así, el número tiene un factor entero y no es primo. (Este no es un método riguroso o eficiente para encontrar números primos, pero funciona como un ejemplo simple). Las estructuras de controles de bucle son más adecuadas para tales casos, en particular, la instrucción `while`:

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// The first factor to try
let factor = 2;

// Execute the block statement if factor is
// less than candidate and keep doing it
// while factor is less than candidate
while ( factor < candidate )
{

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }

  // The next factor to try. Simply
  // increment the current factor by one
  factor++;
}

// Display the result in the console window.
// If candidate has no integer factor, then
// the auxiliary variable is_prime still true
if ( is_prime )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

La declaración de bloque que sigue a la instrucción `while` se ejecutará repetidamente siempre que la condición `factor < candidate` sea verdadera. Se ejecutará al menos una vez, siempre que

inicialicemos la variable `factor` con un valor menor que `candidate`. La estructura `if` anidada en la estructura `while` evaluará si el resto de `candidate` dividido por `factor` es cero. Si es así, el número de `candidate` no es primo y el ciclo puede terminar. La instrucción `break` finalizará el ciclo y la ejecución saltará a la primera instrucción después del bloque `while`.

Tenga en cuenta que el resultado de la condición utilizada por la instrucción `while` debe cambiar en cada bucle, de lo contrario, la instrucción de bloque se repetirá "para siempre". En el ejemplo, incrementamos la variable `factor` –el siguiente divisor que queremos probar– y garantiza que el ciclo terminará en algún punto.

Esta sencilla implementación de comprobación de números primos funciona como se esperaba. Sin embargo, sabemos que un número que no es divisible por dos no será divisible por ningún otro número par. Por lo tanto, podríamos simplemente omitir los números pares agregando otra instrucción `if`:

```
while ( factor < candidate )
{

  // Skip even factors bigger than two
  if ( factor > 2 && factor % 2 == 0 )
  {
    factor++;
    continue;
  }

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }

  // The next number that will divide the candidate
  factor++;
}
```

La sentencia `continue` es similar a la sentencia `break`, pero en lugar de terminar esta iteración del ciclo, ignorará el resto del bloque del ciclo y comenzará una nueva iteración. Tenga en cuenta que la variable `factor` se modificó antes de la declaración `continue`, de lo contrario, el ciclo volvería a tener el mismo resultado en la siguiente iteración. Este ejemplo es demasiado simple y omitir parte del ciclo no mejorará realmente su rendimiento, pero omitir instrucciones redundantes es muy importante al escribir aplicaciones eficientes.

Los bucles se utilizan con tanta frecuencia que existen en muchas variantes diferentes. El ciclo `for` es especialmente adecuado para iterar a través de valores secuenciales, porque nos permite definir las reglas del ciclo en una sola línea:

```
for ( let factor = 2; factor < candidate; factor++ )
{
  // Skip even factors bigger than two
  if ( factor > 2 && factor % 2 == 0 )
  {
    continue;
  }

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }
}
```

Este ejemplo produce exactamente el mismo resultado que el ejemplo anterior `while`, pero su expresión entre paréntesis incluye tres partes, separadas por punto y coma: la inicialización (`let factor = 2`), la condición de bucle (`factor < candidate`) y la expresión final que se evaluará al final de cada iteración del ciclo (`factor++`). Las declaraciones `continue` y `break` también se aplican a los bucles `for`. La expresión final entre paréntesis (`factor++`) se evaluará después de la instrucción `continue`, por lo que no debería estar dentro de la instrucción del bloque, de lo contrario se incrementará dos veces antes de la siguiente iteración.

JavaScript tiene tipos especiales de bucles `for` para trabajar con objetos similares a matrices. Podríamos, por ejemplo, verificar una matriz de variables candidatas en lugar de solo una:

```
// A naive prime number tester

// The array of numbers we want to evaluate
let candidates = [111, 139, 293, 327];

// Evaluates every candidate in the array
for (candidate of candidates)
{
  // Auxiliary variable
  let is_prime = true;

  for ( let factor = 2; factor < candidate; factor++ )
  {
    // Skip even factors bigger than two
    if ( factor > 2 && factor % 2 == 0 )
    {
      continue;
    }

    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime
      is_prime = false;
      break;
    }
  }

  // Display the result in the console window
  if ( is_prime )
  {
    console.log(candidate, "is prime");
  }
  else
  {
    console.log(candidate, "is not prime");
  }
}
```

La declaración `for (candidate of candidates)` asigna un elemento de la matriz `candidates` a la variable `candidate` y lo usa en la declaración de bloque, repitiendo el proceso para cada elemento de la matriz. No es necesario declarar `candidate` por separado, porque el ciclo `for` lo define. Finalmente, el mismo código del ejemplo anterior fue anidado en esta nueva declaración de bloque, pero esta vez probando cada candidato en la matriz.

## Ejercicios guiados

1. ¿Qué valores de la variable `my_var` coinciden con la condición `my_var > 0 && my_var < 9`?

2. ¿Qué valores de la variable `my_var` coinciden con la condición `my_var > 0 || my_var < 9`?

3. ¿Cuántas veces ejecuta el siguiente bucle `while` su declaración de bloque?

```
let i = 0;
while ( 1 )
{
  if ( i == 10 )
  {
    continue;
  }
  i++;
}
```

## Ejercicios de exploración

1. ¿Qué sucede si se usa el operador de asignación igual `=` en lugar del operador de comparación igual `==`?

2. Escriba un fragmento de código usando la estructura de control `if` donde una comparación de igualdad ordinaria devolverá verdadero, pero una comparación de igualdad estricta no lo hará.

3. Reescriba la siguiente instrucción `for` utilizando el operador lógico unario NOT en la condición de bucle. El resultado de la condición debe ser el mismo.

```
for ( let factor = 2; factor < candidate; factor++ )
```

4. Basado en los ejemplos de esta lección, escriba una estructura de control de bucle que imprima todos los factores enteros de un número dado.

## Resumen

Esta lección cubre cómo usar estructuras de control en código JavaScript. Las estructuras condicionales y de bucle son elementos esenciales de cualquier paradigma de programación, y el desarrollo web JavaScript no es una excepción. La lección abarca los siguientes conceptos y procedimientos:

- La instrucción `if` y los operadores de comparación.
- Cómo usar la estructura `switch` con `case`, `default` y `break`.
- La diferencia entre comparación ordinaria y estricta.
- Estructuras de control de bucle: `while` y `for`.

## Respuestas a los ejercicios guiados

1. ¿Qué valores de la variable `my_var` coinciden con la condición `my_var > 0 && my_var < 9`?

   Sólo números que son tanto mayores que 0 como menores que 9. El operador lógico `&&` (AND) requiere que ambas comparaciones coincidan.

2. ¿Qué valores de la variable `my_var` coinciden con la condición `my_var > 0 || my_var < 9`?

   El uso del operador lógico `||` (OR) hará que cualquier número coincida, ya que cualquier número será mayor que 0 o menor que 9.

3. ¿Cuántas veces ejecuta el siguiente bucle `while` su declaración de bloque?

```
let i = 0;
while ( 1 )
{
  if ( i == 10 )
  {
    continue;
  }
  i++;
}
```

   The block statement will repeat indefinitely, as no stop condition was supplied.

## Respuestas a los ejercicios de exploración

1. ¿Qué sucede si se usa el operador de asignación igual `=` en lugar del operador de comparación igual `==`?

   El valor del lado derecho del operador se asigna a la variable de la izquierda y el resultado se pasa a la comparación, que puede no ser el comportamiento deseado.

2. Escriba un fragmento de código usando la estructura de control `if` donde una comparación de igualdad ordinaria devolverá verdadero, pero una comparación de igualdad estricta no lo hará.

```
let a = "1";
let b = 1;

if ( a == b )
{
  console.log("An ordinary comparison will match.");
}

if ( a === b )
{
  console.log("A strict comparison will not match.");
}
```

3. Reescriba la siguiente instrucción `for` utilizando el operador lógico unario NOT en la condición de bucle. El resultado de la condición debe ser el mismo.

```
for ( let factor = 2; factor < candidate; factor++ )
```

   Answer:

```
for ( let factor = 2; ! (factor >= candidate); factor++ )
```

4. Basándose en los ejemplos de esta lección, escriba una estructura de control de bucle que imprima todos los factores enteros de un número dado.

```
for ( let factor = 2; factor <= my_number; factor++ )
{
  if ( my_number % factor == 0 )
  {
    console.log(factor, " is an integer factor of ", my_number);
  }
}
```




# 034.3 Lección 2

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 034 Programación JavaScript |
| **Objetivo:** | 034.3 Funciones y estructuras de control en JavaScript |
| **Lección:** | 2 de 2 |

## Introducción

Además del conjunto estándar de funciones integradas que proporciona el lenguaje JavaScript, los desarrolladores pueden escribir sus propias funciones personalizadas para asignar una entrada a una salida adecuada a las necesidades de la aplicación. Las funciones personalizadas son básicamente un conjunto de declaraciones encapsuladas para usarse en otros lugares como parte de una expresión.

El uso de funciones es una buena forma de evitar escribir código duplicado, ya que se pueden llamar desde diferentes ubicaciones a lo largo del programa. Además, agrupar declaraciones en funciones facilita la vinculación de acciones personalizadas a eventos, que es un aspecto central de la programación de JavaScript.

### Definición de una función

A medida que un programa crece, se vuelve más difícil organizar lo que hace sin utilizar funciones. Cada función tiene su propio alcance de variable privada, por lo que las variables definidas dentro de una función estarán disponibles solo dentro de esa misma función. Por lo tanto, no se mezclarán con variables de otras funciones. Las variables globales todavía son accesibles desde dentro de las

funciones, pero la forma preferible de enviar valores de entrada a una función es a través de *parámetros de función*. Como ejemplo, vamos a construir sobre el validador de números primos de la lección anterior:

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// Start with the lowest prime number after 1
let factor = 2;

// Keeps evaluating while factor is less than the candidate
while ( factor < candidate )
{

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }

  // The next number that will divide the candidate
  factor++;
}

// Display the result in the console window
if ( is_prime )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

Si más adelante en el código necesita verificar si un número es primo, sería necesario repetir el código que ya ha sido escrito. No se recomienda esta práctica, ya que cualquier corrección o mejora del código original deberá replicarse manualmente en todos los lugares donde se haya copiado el

código. Además, la repetición de código supone una carga para el navegador y la red, lo que posiblemente ralentice la visualización de la página web. En lugar de hacer esto, mueva las declaraciones apropiadas a una función:

```
// A naive prime number tester function
function test_prime(candidate)
{
  // Auxiliary variable
  let is_prime = true;

  // Start with the lowest prime number after 1
  let factor = 2;

  // Keeps evaluating while factor is less than the candidate
  while ( factor < candidate )
  {

    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime
      is_prime = false;
      break;
    }

    // The next number that will divide the candidate
    factor++;
  }

  // Send the answer back
  return is_prime;
}
```

La declaración de la función comienza con la palabra reservada `function`, seguida del nombre de la función y sus parámetros. El nombre de la función debe seguir las mismas reglas que los nombres de las variables. Los parámetros de la función, también conocidos como los *argumentos*, están separados por comas y entre paréntesis.

**TIP** No es obligatorio especificar los argumentos en la declaración de la función. Los argumentos pasados a una función se pueden recuperar de un objeto similar a un arreglo dentro de esa función. El índice de los argumentos comienza en 0, por lo que el primer argumento es `arguments[0]`, el segundo argumento es `arguments[1]`, y así sucesivamente.

En el ejemplo, la función `test_prime` tiene un solo argumento: el argumento `candidate`, que es el número primo candidato a ser probado. Los argumentos de función actúan como variables, pero sus valores son asignados por la declaración que llama a la función. Por ejemplo, la instrucción `test_prime (231)` llamará a la función `test_prime` y asignará el valor 231 al argumento `candidate`, que luego estará disponible dentro del cuerpo de la función como una variable ordinaria.

Si la declaración de llamada usa variables simples para los parámetros de la función, sus valores se copiarán a los argumentos de la función. Este procedimiento — copiar los valores de los parámetros usados en la declaración de llamada a los parámetros usados dentro de la función — se llama *pasar argumentos por valor*. Cualquier modificación realizada al argumento por la función no afecta la variable original utilizada en la declaración de llamada. Sin embargo, si la declaración de llamada usa objetos complejos como argumentos (es decir, un objeto con propiedades y métodos adjuntos) para los parámetros de la función, serán *pasados como referencia* y la función podrá modificar el objeto original usado en la declaración de llamada.

Los argumentos que se pasan por valor, así como las variables declaradas dentro de la función, no son visibles fuera de ella. Es decir, su alcance está restringido al cuerpo de la función donde fueron declarados. No obstante, las funciones se emplean generalmente para crear una salida visible fuera de la función. Para compartir un valor con su función de llamada, una función define una declaración `return`.

Por ejemplo, la función `test_prime` en el ejemplo anterior devuelve el valor de la variable `is_prime`. Por lo tanto, la función puede reemplazar la variable en cualquier lugar donde se usaría en el ejemplo original:

```
// The number we want to evaluate
let candidate = 231;

// Display the result in the console window
if ( test_prime(candidate) )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

La declaración `return`, como su nombre lo indica, devuelve el control a la función de llamada. Por lo tanto, donde quiera que se coloque la instrucción `return` en la función, no se ejecutará nada a continuación. Una función puede contener múltiples declaraciones de retorno. Esta práctica puede

ser útil si algunos están dentro de bloques condicionales de declaraciones, de modo que la función pueda ejecutar o no una instrucción `return` particular en cada ejecución.

Es posible que algunas funciones no devuelvan un valor, por lo que la instrucción `return` no es obligatoria. Las declaraciones internas de la función se ejecutan independientemente de su presencia, por lo que las funciones también se pueden usar para cambiar los valores de las variables globales o el contenido de los objetos pasados por referencia, por ejemplo. No obstante, si la función no tiene una declaración `return`, su valor de retorno predeterminado se establece en `undefined`: una variable reservada que no tiene un valor y no se puede escribir.

### Expresiones de funciones

En JavaScript, las funciones son solo otro tipo de *objeto*. Por lo tanto, las funciones se pueden emplear en el script como variables. Esta característica se vuelve explícita cuando la función se declara usando una sintaxis alternativa, llamada *expresiones de función*:

```
let test_prime = function(candidate)
{
  // Auxiliary variable
  let is_prime = true;

  // Start with the lowest prime number after 1
  let factor = 2;

  // Keeps evaluating while factor is less than the candidate
  while ( factor < candidate )
  {

    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime
      is_prime = false;
      break;
    }

    // The next number that will divide the candidate
    factor++;
  }

  // Send the answer back
  return is_prime;
}
```

La única diferencia entre este ejemplo y la declaración de función en el ejemplo anterior está en la primera línea: `let test_prime = function(candidate)` en lugar de `function test_prime(candidate)`. En una expresión de función, el nombre `test_prime` se usa para el objeto que contiene la función y no para nombrar la función en sí. Las funciones definidas en expresiones de función se llaman de la misma forma que las funciones definidas mediante la sintaxis de declaración. Sin embargo, mientras que las funciones declaradas se pueden llamar antes o después de su declaración, las expresiones de función solo se pueden llamar después de su inicialización. Al igual que con las variables, llamar a una función definida en una expresión antes de su inicialización provocará un error de referencia.

## Funciones Recursivas

Además de ejecutar declaraciones y llamar a funciones integradas, las funciones personalizadas también pueden llamar a otras funciones personalizadas, incluidas ellas mismas. Llamar a una función desde sí misma se llama *función recursiva*. Dependiendo del tipo de problema que intente resolver, el uso de funciones recursivas puede ser más sencillo que el uso de bucles anidados para realizar tareas repetitivas.

Hasta ahora, sabemos cómo usar una función para probar si un número dado es primo. Ahora suponga que desea encontrar el siguiente primo que sigue a un número dado. Puede emplear un ciclo `while` para incrementar el número de `candidate` y escribir un ciclo anidado que buscará factores enteros para ese candidato:

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
  // We are only interested in the positive primes,
  // so we will consider the number 2 as the next
  // prime after any number less than two.
  if ( from < 2 )
  {
    return 2;
  }

  // The number 2 is the only even positive prime,
  // so it will be easier to treat it separately.
  if ( from == 2 )
  {
    return 3;
  }
```

```
  // Decrement "from" if it is an even number
  if ( from % 2 == 0 )
  {
    from--;
  }

  // Start searching for primes greater then 3.

  // The prime candidate is the next odd number
  let candidate = from + 2;

  // "true" keeps the loop going until a prime is found
  while ( true )
  {
    // Auxiliary control variable
    let is_prime = true;

    // "candidate" is an odd number, so the loop will
    // try only the odd factors, starting with 3
    for ( let factor = 3; factor < candidate; factor = factor + 2 )
    {
      if ( candidate % factor == 0 )
      {
        // The remainder is zero, so the candidate is not prime.
        // Test the next candidate
        is_prime = false;
        break;
      }
    }
    // End loop and return candidate if it is prime
    if ( is_prime )
    {
      return candidate;
    }
    // If prime not found yet, try the next odd number
    candidate = candidate + 2;
  }
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));
```

Tenga en cuenta que necesitamos usar una condición constante para el ciclo `while` (la expresión `true` dentro del paréntesis) y la variable auxiliar `is_prime` para saber cuándo detener el ciclo. Aunque esta

solución es correcta, usar bucles anidados no es tan elegante como usar la recursividad para realizar la misma tarea:

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
  // We are only interested in the positive primes,
  // so we will consider the number 2 as the next
  // prime after any number less than two.
  if ( from < 2 )
  {
    return 2;
  }

  // The number 2 is the only even positive prime,
  // so it will be easier to treat it separately.
  if ( from == 2 )
  {
    return 3;
  }

  // Decrement "from" if it is an even number
  if ( from % 2 == 0 )
  {
    from--;
  }

  // Start searching for primes greater then 3.

  // The prime candidate is the next odd number
  let candidate = from + 2;

  // "candidate" is an odd number, so the loop will
  // try only the odd factors, starting with 3
  for ( let factor = 3; factor < candidate; factor = factor + 2 )
  {
    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime.
      // Call the next_prime function recursively, this time
      // using the failed candidate as the argument.
      return next_prime(candidate);
    }
```

```
  }

  // "candidate" is not divisible by any integer factor other
  // than 1 and itself, therefore it is a prime number.
  return candidate;
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));
```

Ambas versiones de `next_prime` devuelven el siguiente número primo después del número dado como único argumento (`from`). La versión recursiva, como la versión anterior, comienza marcando los casos especiales (es decir, números menores o iguales a dos). Luego incrementa el candidato y comienza a buscar cualquier factor entero con el ciclo `for` (observe que el ciclo `while` ya no existe). En ese punto, ya se ha probado el único número primo par, por lo que el candidato y sus posibles factores se incrementan en dos. (Un número impar más dos es el siguiente número impar).

Solo hay dos formas de salir del ciclo `for` en el ejemplo. Si se prueban todos los factores posibles y ninguno de ellos tiene un resto igual a cero al dividir `candidate`, el ciclo `for` se completa y la función devuelve a `candidate` como el siguiente número primo después de `from`. De lo contrario, si `factor` es un factor entero de `candidate` (`candidate % factor == 0`), el valor devuelto proviene de la función `next_prime` llamada recursivamente, esta vez con `candidate` incrementado como su paramétro `from`. Las llamadas para `next_prime` se apilarán una encima de la otra, hasta que un candidato finalmente no muestre factores enteros. Luego, la última instancia de `next_prime` que contiene el número primo lo devolverá a la instancia anterior de `next_prime`, y así sucesivamente hasta la primera instancia de `next_prime`. Aunque cada invocación de la función utiliza los mismos nombres para las variables, las invocaciones están aisladas entre sí, por lo que sus variables se mantienen separadas en la memoria.

## Ejercicios guiados

1. ¿Qué tipo de sobrecarga pueden mitigar los desarrolladores mediante el uso de funciones?

2. ¿Cuál es la diferencia entre los argumentos de función pasados por valor y los argumentos de función pasados por referencia?

3. ¿Qué valor se utilizará como resultado de una función personalizada si no tiene una declaración de retorno?

## Ejercicios de exploración

1. ¿Cuál es la causa probable de un *Error de referencia no capturado* emitido al llamar a una función declarada con la sintaxis *expresión*?

2. Escriba una función llamada `multiples_of` que reciba tres argumentos: `factor`, `from` y `to`. Dentro de la función, use la instrucción `console.log ()` para imprimir todos los múltiplos de `factor` que se encuentran entre `from` y `to`.

## Resumen

Esta lección cubre cómo escribir funciones personalizadas en código JavaScript. Las funciones personalizadas permiten al desarrollador dividir la aplicación en “fragmentos” de código reutilizables, lo que facilita la escritura y el mantenimiento de programas más grandes. La lección abarca los siguientes conceptos y procedimientos:

- Cómo definir una función personalizada: declaraciones de funciones y expresiones de funciones.
- Uso de parámetros como entrada de función.
- Uso de la instrucción `return` para establecer la salida de la función.
- Función recursiva.

## Respuestas a los ejercicios guiados

1. ¿Qué tipo de sobrecarga pueden mitigar los desarrolladores mediante el uso de funciones?

   Las funciones nos permiten reutilizar código, lo que facilita su mantenimiento. Un archivo de secuencia de comandos más pequeño también ahorra memoria y tiempo de descarga.

2. ¿Cuál es la diferencia entre los argumentos de función pasados por valor y los argumentos de función pasados por referencia?

   Cuando se pasa por valor, el argumento se copia a la función y la función no puede modificar la variable original en la declaración de llamada. Cuando se pasa por referencia, la función puede manipular la variable original utilizada en la declaración de llamada.

3. ¿Qué valor se utilizará como resultado de una función personalizada si no tiene una declaración de retorno?

   El valor devuelto se establecerá en `undefined`.

## Respuestas a los ejercicios de exploración

1. ¿Cuál es la causa probable de un *Error de referencia no capturado* emitido al llamar a una función declarada con la sintaxis *expresión*?

   La función se llamó antes de su declaración en el archivo de secuencia de comandos.

2. Escriba una función llamada `multiples_of` que reciba tres argumentos: `factor`, `from` y `to`. Dentro de la función, usa la instrucción `console.log()` para imprimir todos los múltiplos de `factor` que se encuentran entre `from` y `to`.

```
function multiples_of(factor, from, to)
{
  for ( let number = from; number <= to; number++ )
  {
    if ( number % factor == 0 )
    {
      console.log(factor, "*", number / factor, "=", number);
    }
  }
}
```




# 034.4 Manipulación JavaScript del contenido y estilo del sitio web

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 034.4

**Peso**

4

**Áreas de conocimiento clave**

- Comprender los valores verdaderos y falsos.
- Comprender los operadores de comparación
- Comprender la diferencia entre comparación flexible y estricta
- Usar condicionales
- Usar bucles
- Definir funciones personalizadas

**Lista parcial de archivos, términos y utilidades**

- `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()`, `document.querySelectorAll()`
- `innerHTML`, `setAttribute()`, `removeAttribute()` propiedades y métodos de los elementos DOM
- `classList`, `classList.add()`, `classList.remove()`, `classList.toggle()` propiedades y métodos de los elementos DOM
- `onClick`, `onMouseOver`, `onMouseOut` atributos de elementos HTML




# 034.4 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 034 Programación JavaScript |
| **Objetivo:** | 034.4 Manipulación JavaScript del contenido y estilo del sitio web |
| **Lección:** | 1 de 1 |

## Introducción

HTML, CSS y JavaScript son tres tecnologías distintas que se unen en la Web. Para crear páginas verdaderamente dinámicas e interactivas, el programador de JavaScript debe combinar componentes de HTML y CSS en tiempo de ejecución, una tarea que se facilita enormemente mediante el uso del *Document Object Model* (DOM).

### Interactuar con el DOM

El DOM es una estructura de datos que funciona como una interfaz de programación para el documento, donde cada aspecto del documento se representa como un nodo en el DOM y cada cambio realizado en el DOM repercutirá inmediatamente en el documento. Para mostrar cómo usar DOM en JavaScript, guarde el siguiente código HTML en un archivo llamado `example.html`:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first">
<p>The dynamic content goes here</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section</p>
</div><!-- #content_second -->

</body>
</html>
```

El DOM estará disponible solo después de que se cargue el HTML, así que escriba el siguiente código JavaScript al final del cuerpo de la página (antes de la etiqueta final `</body>`):

```
<script>
let body = document.getElementsByTagName("body")[0];
console.log(body.innerHTML);
</script>
```

El objeto `document` es el elemento DOM superior, todos los demás elementos se derivan de él. El método `getElementsByTagName()` lista todos los elementos que descienden de `document` que tienen el nombre de etiqueta dado. Aunque la etiqueta `body` se usa solo una vez en el documento, el método `getElementsByTagName()` siempre devuelve una colección similar a un arreglo de elementos encontrados, de ahí el uso del índice `[0]` para devolver el primero (y único) elemento encontrado.

## Contenido HTML

Como se muestra en el ejemplo anterior, el elemento DOM devuelto por `document.getElementsByTagName("body")[0]` se asignó a la variable `body`. La variable `body` se puede usar para manipular el elemento del cuerpo de la página, ya que hereda todos los métodos y atributos DOM de ese elemento. Por ejemplo, la propiedad `innerHTML` contiene el código de marcado HTML completo escrito dentro del elemento correspondiente, por lo que puede usarse para leer el marcado interno. Nuestra llamada `console.log(body.innerHTML)` imprime el contenido dentro de

`<body></body>` en la consola web. La variable también se puede usar para reemplazar ese contenido, como en `body.innerHTML = "<p>Contenido borrado</p>"`.

En lugar de cambiar porciones enteras del marcado HTML, es más práctico mantener inalterada la estructura del documento y simplemente interactuar con sus elementos. Una vez que el navegador representa el documento, todos los elementos son accesibles mediante métodos DOM. Es posible, por ejemplo, listar y acceder a todos los elementos HTML usando la cadena especial `*` en el método `getElementsByTagName()` del objeto `document`:

```
let elements = document.getElementsByTagName("*");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

Este código colocará todos los elementos que se encuentran en `document` en la variable `elements`. La variable `elements` es un objeto similar a un arreglo, por lo que podemos iterar a través de cada uno de sus elementos con un ciclo `for`. Si la página HTML donde se ejecuta este código tiene un elemento con un atributo `id` establecido en `content_first` (consulte la página HTML de muestra que se muestra al comienzo de la lección), la declaración `if` coincide con ese elemento y su contenido de marcado se cambia a `<p>New content</p>`. Tenga en cuenta que se puede acceder a los atributos de un elemento HTML en el DOM utilizando la notación *dot* de las propiedades del objeto JavaScript: por lo tanto, `element.id` se refiere al atributo `id` del elemento actual del bucle `for`. El método `getAttribute()` también podría usarse, como en `element.getAttribute("id")`.

No es necesario recorrer todos los elementos si desea inspeccionar solo un subconjunto de ellos. Por ejemplo, el método `document.getElementsByClassName()` limita los elementos coincidentes a aquellos que tienen una clase específica:

```
let elements = document.getElementsByClassName("content");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

Sin embargo, iterar a través de muchos elementos del documento usando un bucle no es la mejor estrategia cuando tiene que cambiar un elemento específico en la página.

## Seleccionar elementos específicos

JavaScript proporciona métodos optimizados para seleccionar el elemento exacto en el que desea trabajar. El ciclo anterior podría ser reemplazado por completo por el método `document.getElementById()`:

```
let element = document.getElementById("content_first");
element.innerHTML = "<p>New content</p>";
```

Cada atributo `id` en el documento debe ser único, por lo que el método `document.getElementById()` devuelve solo un objeto DOM. Incluso la declaración de la variable `element` se puede omitir, porque JavaScript nos permite encadenar métodos directamente:

```
document.getElementById("content_first").innerHTML = "<p>New content</p>";
```

El método `getElementById()` es el método preferible para localizar elementos en el DOM, porque su rendimiento es mucho mejor que los métodos iterativos cuando se trabaja con documentos complejos. Sin embargo, no todos los elementos tienen un ID explícito, y el método devuelve un valor *null* si ningún elemento coincide con el ID proporcionado (esto también evita el uso de funciones o atributos encadenados, como el `innerHTML` utilizado en el ejemplo anterior). Además, es más práctico asignar atributos de ID solo a los componentes principales de la página y luego usar selectores CSS para ubicar sus elementos secundarios.

Los selectores, presentados en una lección anterior sobre CSS, son patrones que coinciden con elementos del DOM. El método `querySelector()` devuelve el primer elemento coincidente en el árbol del DOM, mientras que `querySelectorAll()` devuelve todos los elementos que coinciden con el selector especificado.

En el ejemplo anterior, el método `getElementById()` selecciona el elemento que lleva el ID `content_first`. El método `querySelector()` puede realizar la misma tarea:

```
document.querySelector("#content_first").innerHTML = "<p>New content</p>";
```

Debido a que el método `querySelector()` usa la sintaxis del selector, la ID proporcionada debe comenzar con un caracter hash. Si no se encuentra ningún elemento coincidente, el método `querySelector()` devuelve *null*.

En el ejemplo anterior, todo el contenido del div `content_first` se reemplaza por la cadena de texto proporcionada. La cadena contiene código HTML, lo que no se considera una práctica recomendada. Debe tener cuidado al agregar marcado HTML codificado de forma rígida al código JavaScript, ya que los elementos de seguimiento pueden resultar difíciles cuando se requieren cambios en la estructura general del documento.

Los selectores no están restringidos al ID del elemento. El elemento interno `p` se puede direccionar directamente:

```
document.querySelector("#content_first p").innerHTML = "New content";
```

El selector `#content_first p` coincidirá solo con el primer elemento `p` dentro del div `#content_first`. Funciona bien si queremos manipular el primer elemento. Sin embargo, es posible que deseemos cambiar el segundo párrafo:

```
<div class="content" id="content_first">
<p>Don't change this paragraph.</p>
<p>The dynamic content goes here.</p>
</div><!-- #content_first -->
```

En este caso, podemos usar la pseudoclase `:nth-child(2)` para que coincida con el segundo elemento `p`:

```
document.querySelector("#content_first p:nth-child(2)").innerHTML = "New content";
```

El número `2` en `p:nth-child(2)` indica el segundo párrafo que coincide con el selector. Vea la lección de selectores CSS para saber más sobre los selectores y cómo usarlos.

## Trabajar con atributos

La capacidad de JavaScript para interactuar con DOM no se limita a la manipulación de contenido. De hecho, el uso más generalizado de JavaScript en el navegador es modificar los atributos de los elementos HTML existentes.

Digamos que nuestra página de ejemplo HTML original ahora tiene tres secciones de contenido:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first" hidden>
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third" hidden>
<p>Third section.</p>
</div><!-- #content_third -->

</body>
</html>
```

Es posible que desee hacer que solo uno de ellos sea visible a la vez, de ahí el atributo `hidden` en todas las etiquetas `div`. Esto es útil, por ejemplo, para mostrar solo una imagen de una galería de imágenes. Para que uno de ellos sea visible cuando se cargue la página, agregue el siguiente código JavaScript a la página:

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
  break;
  case 1:
    content_visible = "#content_second";
  break;
  case 2:
    content_visible = "#content_third";
  break;
}

document.querySelector(content_visible).removeAttribute("hidden");
```

La expresión evaluada por la declaración `switch` devuelve aleatoriamente el número 0, 1 o 2. El selector de ID correspondiente se asigna a la variable `content_visible`, que utiliza el método `querySelector(content_visible)`. La llamada encadenada `removeAttribute("hidden")` elimina el atributo `hidden` del elemento.

El enfoque opuesto también es posible: todas las secciones podrían ser inicialmente visibles (sin el atributo `hidden`) y el programa JavaScript puede asignar el atributo `hidden` a cada sección excepto a la de `content_visible`. Para hacerlo, debe iterar a través de todos los elementos div de contenido que son diferentes del elegido, lo cual se puede hacer usando el método `querySelectorAll()`:

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
  break;
  case 1:
    content_visible = "#content_second";
  break;
  case 2:
    content_visible = "#content_third";
  break;
}

// Hide all content divs, except content_visible
for ( element of document.querySelectorAll(".content:not("+content_visible+")") )
{
  // Hidden is a boolean attribute, so any value will enable it
  element.setAttribute("hidden", "");
}
```

Si la variable `content_visible` se estableció en `#content_first`, el selector será `.content:not(#content_first)`, que se lee como todos los elementos que tienen la clase `content` excepto los que tienen el ID `content_first`. El método `setAttribute()` agrega o cambia atributos de elementos HTML. Su primer parámetro es el nombre del atributo y el segundo es el valor del atributo.

Sin embargo, la forma correcta de cambiar la apariencia de los elementos es con CSS. En este caso, podemos establecer la propiedad CSS `display` en `hidden` y luego cambiarla a `block` usando JavaScript:

```
<style>
div.content { display: none }
</style>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->

<script>
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
  break;
  case 1:
    content_visible = "#content_second";
  break;
  case 2:
    content_visible = "#content_third";
  break;
}
document.querySelector(content_visible).style.display = "block";
</script>
```

Las mismas buenas prácticas que se aplican a la mezcla de etiquetas HTML con JavaScript se aplican también a CSS. Por lo tanto, no se recomienda escribir propiedades CSS directamente en código JavaScript. La forma correcta de alternar el estilo visual es seleccionar una clase CSS predefinida para el elemento.

## Trabajar con clases

Los elementos pueden tener más de una clase asociada, lo que facilita la escritura de estilos que se

pueden agregar o eliminar cuando sea necesario. Sería agotador cambiar muchos atributos CSS directamente en JavaScript, por lo que puede crear una nueva clase CSS con esos atributos y luego agregar la clase al elemento. Los elementos DOM tienen la propiedad `classList`, que se puede usar para ver y manipular las clases asignadas al elemento correspondiente.

Por ejemplo, en lugar de cambiar la visibilidad del elemento, podemos crear una clase CSS adicional para resaltar nuestro div `content`:

```
div.content {
  border: 1px solid black;
  opacity: 0.25;
}
div.content.highlight {
  border: 1px solid red;
  opacity: 1;
}
```

Esta hoja de estilo agregará un borde negro delgado y semitransparencia a todos los elementos que tengan la clase `content`. Solo los elementos que también tienen la clase `highlight` serán completamente opacos y tendrán el borde rojo fino. Entonces, en lugar de cambiar las propiedades CSS directamente como lo hicimos antes, podemos usar el método `classList.add("highlight")` en el elemento seleccionado:

```
// Which content to highlight
let content_highlight;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_highlight = "#content_first";
  break;
  case 1:
    content_highlight = "#content_second";
  break;
  case 2:
    content_highlight = "#content_third";
  break;
}

// Highlight the selected div
document.querySelector(content_highlight).classList.add("highlight");
```

Todas las técnicas y ejemplos que hemos visto hasta ahora se realizaron al final del proceso de carga de la página, pero no se limitan a esta etapa. De hecho, lo que hace que JavaScript sea tan útil para los desarrolladores web es su capacidad para reaccionar a los eventos en la página, que veremos a continuación.

## Controladores de eventos

Todos los elementos visibles de la página son susceptibles a eventos interactivos, como el clic o el movimiento del mouse. Podemos asociar acciones personalizadas a estos eventos, lo que amplía enormemente lo que puede hacer un documento HTML.

Probablemente el elemento HTML más obvio que se beneficia de una acción asociada es el elemento `button`. Para mostrar cómo funciona, agregue tres botones sobre el primer elemento `div` de la página de ejemplo:

```
<p>
<button>First</button>
<button>Second</button>
<button>Third</button>
</p>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->
```

Los botones no hacen nada por sí mismos, pero suponga que desea resaltar el `div` correspondiente al botón presionado. Podemos usar el atributo `onClick` para asociar una acción a cada botón:

```
<p>
<button
onClick="document.getElementById('content_first').classList.toggle('highlight')">Fi
rst</button>
<button
onClick="document.getElementById('content_second').classList.toggle('highlight')">S
econd</button>
<button
onClick="document.getElementById('content_third').classList.toggle('highlight')">Th
ird</button>
</p>
```

El método `classList.toggle()` agrega la clase especificada al elemento si no está presente, y elimina esa clase si ya está presente. Si ejecuta el ejemplo, notará que se puede resaltar más de un `div` al mismo tiempo. Para resaltar solo el `div` correspondiente al botón presionado, es necesario eliminar la clase `highlight` de los otros elementos `div`. No obstante, si la acción personalizada es demasiado larga o involucra más de una línea de código, es más práctico escribir una función aparte de la etiqueta del elemento:

```
function highlight(id)
{
  // Remove the "highlight" class from all content elements
  for ( element of document.querySelectorAll(".content") )
  {
    element.classList.remove('highlight');
  }

  // Add the "highlight" class to the corresponding element
  document.getElementById(id).classList.add('highlight');
}
```

Como en los ejemplos anteriores, esta función se puede colocar dentro de una etiqueta `<script>` o en un archivo JavaScript externo asociado con el documento. La función `highlight` primero elimina la clase `highlight` de todos los elementos `div` asociados con la clase `content`, luego agrega la clase `highlight` al elemento elegido. Cada botón debería llamar a esta función desde su atributo `onClick`, usando el ID correspondiente como argumento de la función:

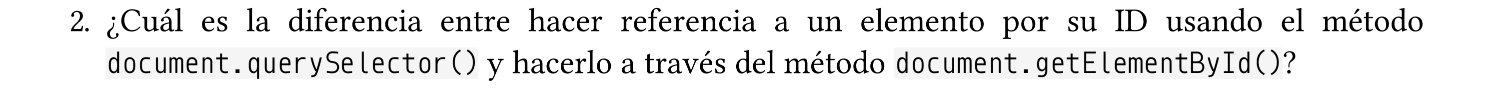
```
<p>
<button onClick="highlight('content_first')">First</button>
<button onClick="highlight('content_second')">Second</button>
<button onClick="highlight('content_third')">Third</button>
</p>
```

Además del atributo `onClick`, podríamos usar el atributo `onMouseOver` (que se activa cuando el dispositivo señalador se usa para mover el cursor sobre el elemento), el atributo `onMouseOut` (que se activa cuando el dispositivo señalador ya no está contenido en el elemento), etc. Además, los controladores de eventos no están restringidos a botones, por lo que puede asignar acciones personalizadas a estos controladores de eventos para todos los elementos HTML visibles.

## Ejercicios guiados

1. Usando el método `document.getElementById()`, ¿cómo podría insertar la frase "Dynamic content" en el contenido interno del elemento cuyo ID es `message`?

2. ¿Cuál es la diferencia entre hacer referencia a un elemento por su ID usando el método `document.querySelector()` y hacerlo a través del método `document.getElementById()`?



3. ¿Cuál es el propósito del método `classList.remove()`?

4. ¿Cuál es el resultado de usar el método `myelement.classList.toggle("active")` si `myelement` no tiene la clase `active` asignada?

## Ejercicios de exploración

1. ¿Qué argumento del método `document.querySelectorAll()` hará que imite el método `document.getElementsByTagName("input")`?

2. ¿Cómo se puede utilizar la propiedad `classList` para listar todas las clases asociadas con un elemento dado?

## Resumen

Esta lección cubre cómo usar JavaScript para cambiar el contenido HTML y sus propiedades CSS usando DOM (Modelo de objetos de documento). Estos cambios pueden ser provocados por eventos de usuario, lo cual es útil para crear interfaces dinámicas. La lección abarca los siguientes conceptos y procedimientos:

- Cómo inspeccionar la estructura del documento usando métodos como `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()` y `document.querySelectorAll()`.
- Cómo cambiar el contenido del documento con la propiedad `innerHTML`.
- Cómo agregar y modificar los atributos de los elementos de la página con los métodos `setAttribute()` y `removeAttribute()`.
- La forma correcta de manipular las clases de elementos usando la propiedad `classList` y su relación con los estilos CSS.
- Cómo vincular funciones a eventos del mouse en elementos específicos.

## Respuestas a los ejercicios guiados

1. Usando el método `document.getElementById()`, ¿cómo podría insertar la frase “Dynamic content” en el contenido interno del elemento cuyo ID es `message`?

   Se puede hacer con la propiedad `innerHTML`:

   ```
   document.getElementById("message").innerHTML = "Dynamic content"
   ```

2. ¿Cuál es la diferencia entre hacer referencia a un elemento por su ID usando el método `document.querySelector()` y hacerlo a través del método `document.getElementById()`?

   La ID debe ir acompañada del caracter hash en las funciones que utilizan selectores, como `document.querySelector()`.

3. ¿Cuál es el propósito del método `classList.remove()`?

   Elimina la clase (cuyo nombre se da como argumento de la función) del atributo `class` del elemento correspondiente.

4. ¿Cuál es el resultado de usar el método `myelement.classList.toggle("active")` si `myelement` no tiene la clase `active` asignada?

   El método asignará la clase `active` a `myelement`.

## Respuestas a los ejercicios de exploración

1. ¿Qué argumento del método `document.querySelectorAll()` hará que imite el método `document.getElementsByTagName("input")`?

   El uso de `document.querySelectorAll("input")` coincidirá con todos los elementos `input` de la página, al igual que `document.getElementsByTagName("input")`.

2. ¿Cómo se puede utilizar la propiedad `classList` para listar todas las clases asociadas con un elemento dado?

   La propiedad `classList` es un objeto similar a un arreglo, por lo que se puede usar un bucle `for` para recorrer todas las clases que contiene.




# Tema 035: Programación NodeJS server



# 035.1 Conceptos básicos de Node.js

### Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 035.1

### Peso

1

### Áreas de conocimiento clave

- Comprender los conceptos de Node.js
- Ejecutar una aplicación NodeJS
- Instalar paquetes NPM

### Lista parcial de archivos, términos y utilidades

- `node [file.js]`
- `npm init`
- `npm install [module_name]`
- `package.json`
- `node_modules`




# 035.1 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 035 Programación NodeJS server |
| **Objetivo:** | 035.1 Conceptos básicos de Node.js |
| **Lección:** | 1 de 1 |

## Introducción

Node.js es un entorno de tiempo de ejecución de JavaScript que ejecuta código JavaScript en servidores web, el llamado *backend* web (lado del servidor), en lugar de usar un segundo lenguaje como Python o Ruby para programas del lado del servidor. El lenguaje JavaScript ya se usa en la parte frontal moderna de las aplicaciones web, interactuando con el HTML y CSS de la interfaz que el usuario interactúa con un navegador web. El uso de Node.js junto con JavaScript en el navegador ofrece la posibilidad de un solo lenguaje de programación para toda la aplicación.

La razón principal de la existencia de Node.js es la forma en que maneja múltiples conexiones concurrentes en el backend. Una de las formas más comunes en que un servidor de aplicaciones web maneja las conexiones es mediante la ejecución de múltiples procesos. Cuando abre una aplicación de escritorio en su computadora, se inicia un proceso y utiliza muchos recursos. Ahora piense en miles de usuarios haciendo lo mismo en una gran aplicación web.

Node.js evita este problema mediante un diseño llamado *event loop*, que es un ciclo interno que comprueba continuamente si se calculan las tareas entrantes. Gracias al uso generalizado de JavaScript y la ubicuidad de las tecnologías web, Node.js ha tenido una gran adopción tanto en aplicaciones pequeñas como grandes. Hay otras características que también ayudaron a que Node.js

se adoptara ampliamente, como el procesamiento de entrada/salida (E/S) asincrónico y sin bloqueo, que se explica más adelante en esta lección.

El entorno Node.js utiliza un motor JavaScript para interpretar y ejecutar código JavaScript en el servidor o en el escritorio. En estas condiciones, el código JavaScript que escribe el programador se analiza y compila justo a tiempo para ejecutar las instrucciones de la máquina generadas por el código JavaScript original.

**NOTE** A medida que avanza en estas lecciones sobre Node.js, puede notar que el JavaScript de Node.js no es exactamente el mismo que el que se ejecuta en el navegador (que sigue a ECMAScript specification), pero es bastante similar.

## Empezando

Esta sección y los siguientes ejemplos asumen que Node.js ya está instalado en su sistema operativo Linux y que el usuario ya tiene habilidades básicas como ejecutar comandos en la terminal.

Para ejecutar los siguientes ejemplos, cree un directorio de trabajo llamado `node_examples`. Abra un indicador de terminal y escriba `node`. Si ha instalado correctamente Node.js, presentará un mensaje `>` donde puede probar los comandos de JavaScript de forma interactiva. Este tipo de entorno se llama REPL, por "leer, evaluar, imprimir y repetir". Escriba la siguiente entrada (o algunas otras declaraciones de JavaScript) en las indicaciones `>`. Presione la tecla Enter después de cada línea, y el entorno REPL devolverá los resultados de sus acciones:

```
> let array = ['a', 'b', 'c', 'd'];
undefined
> array.map( (element, index) => (`Element: ${element} at index: ${index}`));
[
  'Element: a at index: 0',
  'Element: b at index: 1',
  'Element: c at index: 2',
  'Element: d at index: 3'
]
>
```

El fragmento se escribió utilizando la sintaxis ES6, que ofrece una función de mapa para iterar sobre el arreglo e imprimir los resultados utilizando plantillas de cadenas. Puede escribir prácticamente cualquier comando que sea válido. Para salir de la terminal de Node.js, escriba `.exit`, recordando incluir el periodo inicial.

Para scripts y módulos más largos, es más conveniente usar un editor de texto como VS Code, Emacs

o Vim. Puede guardar las dos líneas de código que se acaban de mostrar (con una pequeña modificación) en un archivo llamado `start.js`:

```
let array = ['a', 'b', 'c', 'd'];
array.map( (element, index) => ( console.log(`Element: ${element} at index: ${
index}`)));
```

Luego, puede ejecutar el script desde el shell para producir los mismos resultados que antes:

```
$ node ./start.js
Element: a at index: 0
Element: b at index: 1
Element: c at index: 2
Element: d at index: 3
```

Antes de sumergirnos en más código, vamos a obtener una descripción general de cómo funciona Node.js, utilizando su entorno de ejecución de un solo hilo y el bucle de eventos.

### Bucle de eventos y subproceso único

Es difícil saber cuánto tiempo le tomará a su programa Node.js manejar una solicitud. Algunas solicitudes pueden ser breves, tal vez simplemente recorrer variables en la memoria y devolverlas, mientras que otras pueden requerir actividades que requieren mucho tiempo, como abrir un archivo en el sistema o emitir una consulta a una base de datos y esperar los resultados. ¿Cómo maneja Node.js esta incertidumbre? El ciclo de eventos es la respuesta.

Imagine a un chef realizando múltiples tareas. Hornear un pastel es una tarea que requiere mucho tiempo para que el horno lo cocine. El chef no se queda allí esperando a que el pastel esté listo y luego se pone a preparar un café. En cambio, mientras el horno hornea el pastel, el chef prepara café y otras tareas en paralelo. Pero el cocinero siempre está comprobando si es el momento adecuado para centrarse en una tarea específica (preparar café) o para sacar el pastel del horno.

El ciclo del evento es como el chef que está constantemente al tanto de las actividades que lo rodean. En Node.js, un “event-checker” siempre está buscando operaciones que se hayan completado o estén esperando ser procesadas por el motor de JavaScript.

Con este enfoque, una operación asincrónica y larga no bloquea otras operaciones rápidas posteriores. Esto se debe a que el mecanismo de bucle de eventos siempre verifica si esa tarea larga, como una operación de E/S, ya está hecha. De lo contrario, Node.js puede continuar procesando otras tareas. Una vez que se completa la tarea en segundo plano, se devuelven los resultados y la

aplicación de Node.js puede usar una función de activación (callback) para procesar aún más la salida.

Debido a que Node.js evita el uso de múltiples subprocesos, como hacen otros entornos, se denomina *entorno de un solo subproceso* y, por lo tanto, un enfoque sin bloqueo es de suma importancia. Es por eso que Node.js usa un bucle de eventos. Sin embargo, para las tareas de procesamiento intensivo, Node.js no se encuentra entre las mejores herramientas: existen otros lenguajes y entornos de programación que abordan estos problemas de manera más eficiente.

En las siguientes secciones, veremos más de cerca las funciones de devolución de llamada. Por ahora, comprenda que las funciones de devolución de llamada son activadores que se ejecutan al completar una operación predefinida.

### Módulos

Es una buena práctica dividir la funcionalidad compleja y los fragmentos extensos de código en partes más pequeñas. Hacer esta modularización ayuda a organizar mejor el código base, abstraer las implementaciones y evitar problemas de ingeniería complicados. Para satisfacer esas necesidades, los programadores empaquetan bloques de código fuente para ser consumidos por otras partes internas o externas del código.

Considere el ejemplo de un programa que calcula el volumen de una esfera. Abra su editor de texto y cree un archivo llamado `volumeCalculator.js` que contenga el siguiente JavaScript:

```
const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * radius
}

console.log(`A sphere with radius 3 has a ${sphereVol(3)} volume.`);
console.log(`A sphere with radius 6 has a ${sphereVol(6)} volume.`);
```

Ahora, ejecute el archivo usando Node:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A sphere with radius 6 has a 904.7786842338603 volume.
```

Aquí, se utilizó una función simple para calcular el volumen de una esfera, en función de su radio. Imagine que también necesitamos calcular el volumen de un cilindro, cono, etc.: notamos rápidamente que esas funciones específicas deben agregarse al archivo `volumeCalculator.js`, que

puede convertirse en una enorme colección de funciones. Para organizar mejor la estructura, podemos usar los módulos como paquetes de código separado.

Para hacer eso, cree un archivo separado llamado `polyhedrons.js`:

```
const coneVol = (radius, height) => {
  return 1 / 3 * Math.PI * Math.pow(radius, 2) * height;
}

const cylinderVol = (radius, height) => {
  return Math.PI * Math.pow(radius, 2) * height;
}

const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * Math.pow(radius, 3);
}

module.exports = {
  coneVol,
  cylinderVol,
  sphereVol
}
```

Ahora, en el archivo `volumeCalculator.js`, elimine el código antiguo y reemplácelo con este fragmento:

```
const polyhedrons = require('./polyhedrons.js');

console.log(`A sphere with radius 3 has a ${polyhedrons.sphereVol(3)} volume.`);
console.log(`A cylinder with radius 3 and height 5 has a ${polyhedrons.cylinderVol
(3, 5)} volume.`);
console.log(`A cone with radius 3 and height 5 has a ${polyhedrons.coneVol(3, 5)}
volume.`);
```

Y luego ejecute el nombre del archivo en el entorno Node.js:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A cylinder with radius 3 and height 5 has a 141.3716694115407 volume.
A cone with radius 3 and height 5 has a 47.12388980384689 volume.
```

En el entorno Node.js, cada archivo de código fuente se considera un módulo, pero la palabra "module" en Node.js indica código empaquetado como en el ejemplo anterior. Mediante el uso de módulos, extrajimos las funciones de volumen del archivo principal, `volumeCalculator.js`, reduciendo así su tamaño y facilitando la aplicación de pruebas unitarias, que son una buena práctica al desarrollar aplicaciones del mundo real.

Ahora que sabemos cómo se usan los módulos en Node.js, podemos usar una de las herramientas más importantes: el *Node Package Manager* (NPM).

Uno de los principales trabajos de NPM es administrar, descargar e instalar módulos externos en el proyecto o en el sistema operativo. Puede inicializar un repositorio de nodos con el comando `npm init`.

NPM hará las preguntas predeterminadas sobre el nombre de su repositorio, versión, descripción, etc. Puede omitir estos pasos usando `npm init --yes`, y el comando generará automáticamente un archivo `package.json` que describe las propiedades de su proyecto/módulo.

Abra el archivo `package.json` en su editor de texto favorito y verá un archivo JSON que contiene propiedades como palabras clave, comandos de script para usar con NPM, un nombre, etc.

Una de esas propiedades son las dependencias que están instaladas en su repositorio local. NPM agregará el nombre y la versión de estas dependencias en `package.json`, junto con `package-lock.json`, otro archivo utilizado como respaldo por NPM en caso de que `package.json` falle.

Escriba lo siguiente en su terminal:

```
$ npm i dayjs
```

La bandera `i` es un atajo para el argumento `install`. Si está conectado a Internet, NPM buscará un módulo llamado `dayjs` en el repositorio remoto de Node.js, descargará el módulo e instalará localmente. NPM también agregará esta dependencia a sus archivos `package.json` y `package-lock.json`. Ahora puede ver que hay una carpeta llamada `node_modules`, que contiene el módulo instalado junto con otros módulos si son necesarios. El directorio `node_modules` contiene el código real que se utilizará cuando se importe y se llame a la biblioteca. Sin embargo, esta carpeta no se guarda en los sistemas de control de versiones que utilizan Git, ya que el archivo `package.json` proporciona todas las dependencias utilizadas. Otro usuario puede tomar el archivo `package.json` y simplemente ejecutar `npm install` en su propia máquina, donde NPM creará una carpeta `node_modules` con todas las dependencias del `package.json`, evitando así el control de versiones para el miles de archivos disponibles en el repositorio de NPM.

Ahora que el módulo `dayjs` está instalado en el directorio local, abra la consola de Node.js y escriba

las siguientes líneas:

```
const dayjs = require('dayjs');
dayjs().format('YYYY MM-DDTHH:mm:ss')
```

El módulo `dayjs` se carga con la palabra clave `require`. Cuando se llama a un método del módulo, la biblioteca toma la fecha y hora del sistema actual y la genera en el formato especificado:

```
2020 11-22T11:04:36
```

Este es el mismo mecanismo utilizado en el ejemplo anterior, donde el tiempo de ejecución de Node.js carga la función de terceros en el código.

### Funcionalidad del servidor

Debido a que Node.js controla el back-end de las aplicaciones web, una de sus tareas principales es manejar las solicitudes HTTP.

A continuación, se muestra un resumen de cómo los servidores web manejan las solicitudes HTTP entrantes. La funcionalidad del servidor es escuchar solicitudes, determinar lo más rápido posible qué respuesta necesita cada uno y devolver esa respuesta al remitente de la solicitud. Esta aplicación debe recibir una solicitud HTTP entrante desencadenada por el usuario, analizar la solicitud, realizar el cálculo, generar la respuesta y devolverla. Se utiliza un módulo HTTP como Node.js porque simplifica esos pasos, permitiendo que un programador web se concentre en la aplicación en sí.

Considere el siguiente ejemplo que implementa esta funcionalidad muy básica:

```
const http = require('http');
const url = require('url');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  const queryObject = url.parse(req.url,true).query;
  let result = parseInt(queryObject.a) + parseInt(queryObject.b);

  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Result: ${result}\n`);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Guarde estos contenidos en un archivo llamado `basic_server.js` y ejecútelo a través de un comando `node`. La terminal que ejecuta Node.js mostrará el siguiente mensaje:

```
Server running at http://127.0.0.1:3000/
```

Luego, visite la siguiente URL en su navegador web: `http://127.0.0.1:3000/numbers?a=2&b=17`

Node.js está ejecutando un servidor web en su computadora y usa dos módulos: `http` y `url`. El módulo `http` configura un servidor HTTP básico, procesa las solicitudes web entrantes y las entrega a nuestro código de aplicación simple. El módulo de URL analiza los argumentos pasados en la URL, los convierte a un formato numérico entero (integer) y realiza la operación de suma. El módulo `http` envía la respuesta como texto al navegador web.

En una aplicación web real, Node.js se usa comúnmente para procesar y recuperar datos, generalmente de una base de datos, y devolver la información procesada al front-end para mostrarla. Pero la aplicación básica de esta lección muestra de manera concisa cómo Node.js utiliza módulos para manejar solicitudes web como un servidor web.

## Ejercicios guiados

1. ¿Cuáles son las razones para usar módulos en lugar de escribir funciones simples?

2. ¿Por qué el entorno Node.js se volvió tan popular? Cite una característica.

3. ¿Cuál es el propósito del archivo `package.json`?

4. ¿Por qué no se recomienda guardar y compartir la carpeta `node_modules`?

## Ejercicios Exploratorios

1. ¿Cómo puede ejecutar aplicaciones Node.js en su computadora?

2. ¿Cómo se pueden delimitar los parámetros en la URL para analizar dentro del servidor?

3. Especifique un escenario en el que una tarea específica podría ser un cuello de botella para una aplicación Node.js.

4. ¿Cómo implementaría un parámetro para multiplicar o sumar los dos números en el ejemplo del servidor?

## Resumen

Esta lección proporcionó una descripción general del entorno Node.js, sus características y cómo se puede usar para implementar programas simples. Esta lección incluye los siguientes conceptos:

- Qué es Node.js y por qué se usa.
- Cómo ejecutar programas Node.js usando la línea de comandos.
- El evento `loops` y el hilo único (`single thread`).
- Módulos.
- Node Package Manager (NPM).
- Funcionalidad del servidor.

## Respuestas a los ejercicios guiados

1. ¿Cuáles son las razones para usar módulos en lugar de escribir funciones simples?

   Al optar por módulos en lugar de funciones convencionales, el programador crea una base de código más simple para leer y mantener y para la cual escribir pruebas automatizadas.

2. ¿Por qué el entorno Node.js se volvió tan popular? Cite dos características.

   Una de las razones es la flexibilidad del lenguaje JavaScript, que ya se usaba ampliamente en el front-end de las aplicaciones web. Node.js permite el uso de un solo lenguaje de programación en todo el sistema.

3. ¿Cuál es el propósito del archivo `package.json`?

   Este archivo contiene metadatos para el proyecto, como el nombre, la versión, las dependencias (bibliotecas), etc. Dado un archivo `package.json`, otras personas pueden descargar e instalar las mismas bibliotecas y ejecutar pruebas de la misma manera que lo hizo el creador original.

4. ¿Por qué no se recomienda guardar y compartir la carpeta `node_modules`?

   La carpeta `node_modules` contiene las implementaciones de bibliotecas disponibles en repositorios remotos. Entonces, la mejor manera de compartir estas bibliotecas es indicándolas en el archivo `package.json` y luego usar NPM para descargar esas bibliotecas. Este método es más simple y sin errores, ya que no es necesario realizar un seguimiento ni mantener las bibliotecas localmente.

## Respuestas a los ejercicios de exploración

1. ¿Cómo puede ejecutar aplicaciones Node.js en su computadora?

   Puede ejecutarlos escribiendo `node PATH/FILE_NAME.js` en la línea de comandos de su terminal, cambiando `PATH` por la ruta de su archivo Node.js y cambiando `FILE_NAME.js` por el nombre de archivo elegido.

2. ¿Cómo se pueden delimitar los parámetros en la URL para analizar dentro del servidor?

   El caracter `&` se utiliza para delimitar esos parámetros, de modo que se puedan extraer y analizar en el código JavaScript.

3. Especifique un escenario en el que una tarea específica podría ser un cuello de botella para una aplicación Node.js.

   Node.js no es un buen entorno para ejecutar procesos intensivos de CPU porque utiliza un solo hilo. Un escenario de cálculo numérico podría ralentizar y bloquear toda la aplicación. Si se necesita una simulación numérica, es mejor utilizar otras herramientas.

4. ¿Cómo implementaría un parámetro para multiplicar o sumar los dos números en el ejemplo del servidor?

   Use un operador ternario o una condición if-else para verificar un parámetro adicional. Si el parámetro es la cadena `mult` devuelve el producto de los números, de lo contrario devuelve la suma. Reemplace el código anterior con el fragmento a continuación. Reinicie el servidor en la línea de comandos presionando `Ctrl` + `C` y vuelva a ejecutar el comando para reiniciar el servidor. Ahora pruebe la nueva aplicación visitando la URL `http://127.0.0.1:3000/numbers?a=2&b=17&operation=mult` en su navegador. Si omite o cambia el último parámetro, los resultados deben ser la suma de los números.

   ```
   let result = queryObject.operation == 'mult' ? parseInt(queryObject.a) *
   parseInt(queryObject.b) : parseInt(queryObject.a) + parseInt(queryObject.b);
   ```




# 035.2 Conceptos básicos de NodeJS Express

### Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 035.2

### Peso

4

### Áreas de conocimiento clave

- Definir rutas a archivos estáticos y plantillas EJS
- Servir archivos estáticos a través de Express
- Servir plantillas EJS a través de Express
- Cree plantillas EJS simples y no anidadas
- Use el objeto de solicitud para acceder a los parámetros HTTP GET y POST y procesar los datos enviados a través de formularios HTML
- Conciencia de la validación de entrada del usuario
- Conciencia de Cross-site Scripting (XSS)
- Conciencia de la falsificación de solicitudes entre sitios (CSRF)

### Lista parcial de archivos, términos y utilidades

- Módulos `express` y `body-parser`
- Objecto Express app
- `app.get()`, `app.post()`
- `res.query()`, `res.body()`
- Módulo `ejs`

- `res.render()`
- `<% … %>`, `<%= … %>`, `<%# … %>`, `<%- … %>`
- `views/`




# 035.2 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 035 Programación NodeJS server |
| **Objetivo:** | 035.2 Conceptos básicos de NodeJS Express |
| **Lección:** | 1 de 2 |

## Introducción

Express.js, o simplemente Express, es un marco popular que se ejecuta en Node.js y se usa para escribir servidores HTTP que manejan solicitudes de clientes de aplicaciones web. Express admite muchas formas de leer los parámetros enviados a través de HTTP.

### Script de servidor inicial

Para demostrar las funciones básicas de Express para recibir y manejar solicitudes, simulemos una aplicación que solicita información del servidor. En particular, el servidor de ejemplo:

- Proporciona una función `echo`, que simplemente devuelve el mensaje enviado por el cliente.
- Le dice al cliente su dirección IP a pedido.
- Utiliza cookies para identificar clientes conocidos.

El primer paso es crear el archivo JavaScript que funcionará como servidor. Usando `npm`, cree un directorio llamado `myserver` con el archivo JavaScript:

```
$ mkdir myserver
$ cd myserver/
$ npm init
```

Para el punto de entrada, se puede utilizar cualquier nombre de archivo. Aquí usaremos el nombre de archivo predeterminado: `index.js`. La siguiente lista muestra un archivo básico `index.js` que se utilizará como punto de entrada para nuestro servidor:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.get('/', (req, res) => {
  res.send('Request received')
})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Algunas constantes importantes para la configuración del servidor se definen en las primeras líneas del script. Los dos primeros, `express` y `app`, corresponden al módulo `express` incluido y una instancia de este módulo que ejecuta nuestra aplicación. Agregaremos las acciones a realizar por el servidor al objeto `app`.

Las otras dos constantes, `host` y `port`, definen el host y el puerto de comunicación asociados al servidor.

Si tiene un host de acceso público, use su nombre en lugar de `myserver` como valor de `host`. Si no proporciona el nombre de host, Express se establecerá de forma predeterminada en `localhost`, la computadora donde se ejecuta la aplicación. En ese caso, ningún cliente externo podrá comunicarse con el programa, lo que puede estar bien para realizar pruebas pero ofrece poco valor en la producción.

Es necesario proporcionar el puerto o el servidor no se iniciará.

Este script adjunta solo dos procedimientos al objeto `app`: the `app.get()` que responde a las solicitudes realizadas por los clientes a través de HTTP `GET`, y la llamada `app.listen()`, que se requiere para activar el servidor y le asigna un host y un puerto.

Para iniciar el servidor, simplemente ejecute el comando `node`, proporcionando el nombre del script como argumento:

```
$ node index.js
```

Tan pronto como aparezca el mensaje `Server ready at http://myserver:8080`, el servidor estará listo para recibir solicitudes de un cliente HTTP. Las solicitudes se pueden realizar desde un navegador en la misma computadora donde se ejecuta el servidor, o desde otra máquina que pueda acceder al servidor.

Todos los detalles de la transacción que veremos aquí se muestran en el navegador si abre una ventana para la consola del desarrollador. Alternativamente, el comando `curl` se puede usar para la comunicación HTTP y le permite inspeccionar los detalles de la conexión más fácilmente. Si no está familiarizado con la línea de comandos de shell, puede crear un formulario HTML para enviar solicitudes a un servidor.

El siguiente ejemplo muestra cómo usar el comando `curl` en la línea de comandos para realizar una solicitud HTTP al servidor recién implementado:

```
$ curl http://myserver:8080 -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 16
< ETag: W/"10-1WVvDtVyAF0vX9evlsFlfiJTT5c"
< Date: Fri, 02 Jul 2021 14:35:11 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Request received
```

La opción `-v` del comando `curl` muestra todos los encabezados de solicitud y respuesta, así como otra

información de depuración. Las líneas que comienzan con `>` indican los encabezados de solicitud enviados por el cliente y las líneas que comienzan con `<` indican los encabezados de respuesta enviados por el servidor. Las líneas que comienzan con `*` son información generada por el propio `curl`. El contenido de la respuesta se muestra solo al final, que en este caso es la línea `Request received`.

La URL del servicio, que en este caso contiene el nombre de host y el puerto del servidor (`http://myserver:8080`), se proporcionaron como argumentos para el comando `curl`. Debido a que no se proporciona ningún directorio o nombre de archivo, estos valores predeterminados son el directorio raíz `/`. La barra diagonal aparece como el archivo de solicitud en la línea `> GET / HTTP/1.1`, seguida en la salida por el nombre de host y el puerto.

Además de mostrar los encabezados de conexión HTTP, el comando `curl` ayuda al desarrollo de la aplicación permitiéndole enviar datos al servidor usando diferentes métodos HTTP y en diferentes formatos. Esta flexibilidad hace que sea más fácil depurar cualquier problema e implementar nuevas funciones en el servidor.

## Rutas

Las solicitudes que el cliente puede hacer al servidor dependen de las *rutas* que se hayan definido en el archivo `index.js`. Una ruta especifica un método HTTP y define un *path* (más precisamente, un URI) que puede ser solicitado por el cliente.

Hasta ahora, el servidor solo tiene una ruta configurada:

```
app.get('/', (req, res) => {
  res.send('Request received')
})
```

Aunque es una ruta muy simple, simplemente devolver un mensaje de texto sin formato al cliente, es suficiente para identificar los componentes más importantes que se utilizan para estructurar la mayoría de las rutas:

- El método HTTP servido por la ruta. En el ejemplo, el método HTTP `GET` se indica mediante la propiedad `get` del objeto `app`.
- El camino servido por la ruta. Cuando el cliente no especifica una ruta para la solicitud, el servidor usa el directorio raíz, que es el directorio base reservado para que lo use el servidor web. Un ejemplo posterior en este capítulo usa la ruta `/echo`, que corresponde a una solicitud hecha a `myserver:8080/echo`.
- La función ejecutada cuando el servidor recibe una solicitud en esta ruta, generalmente escrita

en forma abreviada como una *flecha* porque la sintaxis `=>` apunta a la definición de la función sin nombre. El parámetro `req` (abreviatura de "request") y el parámetro `res` (abreviatura de "response") brindan detalles sobre la conexión, pasados a la función por la propia instancia de la aplicación.

### Método POST

Para ampliar la funcionalidad de nuestro servidor de prueba, veamos cómo definir una ruta para el método HTTP `POST`. Los clientes lo utilizan cuando necesitan enviar datos adicionales al servidor además de los incluidos en el encabezado de la solicitud. La opción `--data` del comando `curl` invoca automáticamente el método `POST` e incluye contenido que se enviará al servidor a través de `POST`. La línea `POST / HTTP/1.1` en el siguiente resultado muestra que se utilizó el método `POST`. Sin embargo, nuestro servidor definió solo un método `GET`, por lo que ocurre un error cuando usamos `curl` para enviar una solicitud a través de `POST`:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> POST / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
> Content-Length: 37
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 37 out of 37 bytes
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< X-Powered-By: Express
< Content-Security-Policy: default-src 'none'
< X-Content-Type-Options: nosniff
< Content-Type: text/html; charset=utf-8
< Content-Length: 140
< Date: Sat, 03 Jul 2021 02:22:45 GMT
< Connection: keep-alive
<
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot POST /</pre>
</body>
</html>
* Connection #0 to host myserver left intact
```

En el ejemplo anterior, ejecutar `curl` con el parámetro `--data message="This is the POST request body"` es equivalente a enviar un formulario que contiene el campo de texto llamado `message`, rellenado con `This is the POST request body`.

Debido a que el servidor está configurado con una sola ruta `/`, y esa ruta solo responde al método HTTP `GET`, el encabezado de respuesta tiene la línea `HTTP/1.1 404 Not Found`. Además, Express generó automáticamente una breve respuesta HTML con la advertencia `Cannot POST`.

Habiendo visto cómo generar una solicitud `POST` a través de `curl`, vamos a escribir un programa

Express que pueda manejar la solicitud con éxito.

Primero, tenga en cuenta que el campo `Content-Type` en el encabezado de la solicitud que los datos enviados por el cliente están en el formato `application/x-www-form-urlencoded`. Express no reconoce ese formato por defecto, por lo que necesitamos usar el módulo `express.urlencoded`. Cuando incluimos este módulo, el objeto `req` enviado como parámetro a la función del controlador, tiene la propiedad `req.body.message` establecida, que corresponde al campo `message` enviado por el cliente. El módulo se carga con `app.use`, que debe colocarse antes de la declaración de rutas:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.urlencoded({ extended: true }))
```

Una vez hecho esto, sería suficiente cambiar `app.get` a `app.post` en la ruta existente para cumplir con las solicitudes realizadas a través de `POST` y recuperar el cuerpo de la solicitud:

```
app.post('/', (req, res) => {
  res.send(req.body.message)
})
```

En lugar de reemplazar la ruta, otra posibilidad sería simplemente agregar esta nueva ruta, porque Express identifica el método HTTP en el encabezado de la solicitud y usa la ruta apropiada. Debido a que estamos interesados en agregar más de una funcionalidad a este servidor, es conveniente separar cada una con su propia ruta, como `/echo` y `/ip`.

### Controlador de ruta y función

Habiendo definido qué método HTTP responderá a la solicitud, ahora necesitamos definir una ruta específica para el recurso y una función que procesa y genera una respuesta al cliente.

Para expandir la funcionalidad `echo` del servidor, podemos definir una ruta usando el método `POST` con la ruta `/echo`:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

El parámetro `req` de la función del controlador contiene todos los detalles de la solicitud almacenados como propiedades. El contenido del campo `message` en el cuerpo de la solicitud está disponible en la propiedad `req.body.message`. El ejemplo simplemente envía este campo al cliente a través de la llamada `res.send(req.body.message)`.

Recuerde que los cambios que realice solo tendrán efecto después de reiniciar el servidor. Debido a que está ejecutando el servidor desde una ventana de terminal durante los ejemplos de este capítulo, puede apagar el servidor presionando `Ctrl`+`C` en ese terminal. Luego, vuelva a ejecutar el servidor a través del comando `node index.js`. La respuesta obtenida por el cliente a la solicitud `curl` que mostramos antes ahora es exitosa:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
This is the POST request body
```

#### Otras formas de enviar y recibir información en una solicitud GET

Puede resultar excesivo utilizar el método HTTP `POST` si solo se envían mensajes de texto cortos como el que se utiliza en el ejemplo. En tales casos, los datos se pueden enviar en una *query string* que comienza con un signo de interrogación. Por lo tanto, la cadena `?message=This+is+the+message` podría incluirse dentro de la ruta de solicitud del método HTTP `GET`. Los campos utilizados en la cadena de consulta están disponibles para el servidor en la propiedad `req.query`. Por lo tanto, un campo llamado `message` está disponible en la propiedad `req.query.message`.

Otra forma de enviar datos a través del método HTTP `GET` es usar los *parámetros de ruta* de Express:

```
app.get('/echo/:message', (req, res) => {
  res.send(req.params.message)
})
```

La ruta en este ejemplo coincide con las solicitudes realizadas con el método `GET` usando la ruta `/echo/:message`, donde `:message` es un marcador de posición para cualquier término enviado con esa etiqueta por el cliente. Estos parámetros son accesibles en la propiedad `req.params`. Con esta nueva ruta, el cliente puede solicitar la función `echo` del servidor de manera más precisa:

```
$ curl http://myserver:8080/echo/hello
hello
```

En otras situaciones, la información que el servidor necesita para procesar la solicitud no necesita ser proporcionada explícitamente por el cliente. Por ejemplo, el servidor tiene otra forma de recuperar la

dirección IP pública del cliente. Esa información está presente en el objeto `req` de forma predeterminada, en la propiedad `req.ip`:

```
app.get('/ip', (req, res) => {
  res.send(req.ip)
})
```

Ahora el cliente puede solicitar la ruta `/ip` con el método `GET` para encontrar su propia dirección IP pública:

```
$ curl http://myserver:8080/ip
187.34.178.12
```

El cliente puede modificar otras propiedades del objeto `req`, especialmente los encabezados de solicitud disponibles en `req.headers`. La propiedad `req.headers.user-agent`, por ejemplo, identifica qué programa está realizando la solicitud. Aunque no es una práctica común, el cliente puede cambiar el contenido de este campo, por lo que el servidor no debe usarlo para identificar de manera confiable a un cliente en particular. Es aún más importante validar los datos proporcionados explícitamente por el cliente, para evitar inconsistencias en los límites y formatos que podrían afectar negativamente a la aplicación.

## Ajustes a la respuesta

Como hemos visto en ejemplos anteriores, el parámetro `res` es responsable de devolver una respuesta al cliente. Además, el objeto `res` puede cambiar otros aspectos de la respuesta. Es posible que haya notado que, aunque las respuestas que hemos implementado hasta ahora son breves mensajes de texto sin formato, el encabezado `Content-Type` de las respuestas usa `text/html; charset=utf-8`. Aunque esto no impide que se acepte la respuesta de texto sin formato, será más correcto si redefinimos este campo en el encabezado de la respuesta a `text/plain` con la configuración `res.type('text/plain')`.

Otros tipos de ajustes de respuesta implican el uso de *cookies*, que permiten al servidor identificar a un cliente que ha realizado una solicitud previamente. Las cookies son importantes para funciones avanzadas, como la creación de sesiones privadas que asocian solicitudes a un usuario específico, pero aquí solo veremos un ejemplo simple de cómo usar una cookie para identificar a un cliente que ha accedido previamente al servidor.

Dado el diseño modularizado de Express, la administración de cookies debe instalarse con el comando `npm` antes de usarse en el script:

```
$ npm install cookie-parser
```

Después de la instalación, la administración de cookies debe incluirse en el script del servidor. La siguiente definición debe incluirse cerca del comienzo del archivo:

```
const cookieParser = require('cookie-parser')
app.use(cookieParser())
```

Para ilustrar el uso de cookies, modifiquemos la función del controlador de la ruta con la ruta raíz `/` que ya existe en el script. El objeto `req` tiene una propiedad `req.cookies`, donde se guardan las cookies enviadas en el encabezado de la solicitud. El objeto `res`, por otro lado, tiene un método `res.cookie()` que crea una nueva cookie para ser enviada al cliente. La función de controlador en el siguiente ejemplo verifica si existe una cookie con el nombre `known` en la solicitud. Si dicha cookie no existe, el servidor asume que se trata de un visitante por primera vez y le envía una cookie con ese nombre a través de la llamada `res.cookie('known', '1')`. Asignamos arbitrariamente el valor `1` a la cookie porque se supone que tiene algún contenido, pero el servidor no consulta ese valor. Esta aplicación solo asume que la simple presencia de la cookie indica que el cliente ya ha solicitado esta ruta antes:

```
app.get('/', (req, res) => {
  res.type('text/plain')
  if ( req.cookies.known === undefined ){
    res.cookie('known', '1')
    res.send('Welcome, new visitor!')
  }
  else
    res.send('Welcome back, visitor');
})
```

De forma predeterminada, `curl` no utiliza cookies en las transacciones. Pero tiene opciones para almacenar (`-c cookies.txt`) y enviar cookies almacenadas (`-b cookies.txt`):

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
* Added cookie known="1" for domain myserver, path /, expire 0
< Set-Cookie: known=1; Path=/
< Content-Length: 21
< ETag: W/"15-l7qrxcqicl4xv6EfA5fZFWCFrgY"
< Date: Sat, 03 Jul 2021 23:45:03 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome, new visitor!
```

Debido a que este comando fue el primer acceso desde que se implementaron las cookies en el servidor, el cliente no tenía ninguna cookie para incluir en la solicitud. Como se esperaba, el servidor no identificó la cookie en la solicitud y, por lo tanto, incluyó la cookie en los encabezados de respuesta, como se indica en la línea de la salida del archivo `Set-Cookie: known=1; Path=/`. Dado que hemos habilitado las cookies en `curl`, una nueva solicitud incluirá la cookie `known=1` en los encabezados de la solicitud, lo que permitirá al servidor identificar la presencia de la cookie:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
> Cookie: known=1
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
< Content-Length: 21
< ETag: W/"15-ATq2flQYtLMYIUpJwwpb5SjV9Ww"
< Date: Sat, 03 Jul 2021 23:45:47 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome back, visitor
```

## Seguridad de las cookies

El desarrollador debe ser consciente de las posibles vulnerabilidades al utilizar cookies para identificar a los clientes que realizan solicitudes. Los atacantes pueden utilizar técnicas como *cross-site scripting* (XSS) y *cross-site request forgery* (CSRF) para robar cookies de un cliente y, por lo tanto, hacerse pasar por ellos al realizar una solicitud al servidor. En términos generales, estos tipos de ataques utilizan campos de comentarios no validados o URL construidas meticulosamente para insertar código JavaScript malicioso en la página. Cuando es ejecutado por un cliente auténtico, este código puede copiar cookies válidas y almacenarlas o reenviarlas a otro destino.

Por lo tanto, especialmente en aplicaciones profesionales, es importante instalar y utilizar funciones Express más especializadas, conocidas como *middleware*. El módulo `express-session` o `cookie-session` proporciona un control más completo y seguro sobre la gestión de la sesión y las cookies. Estos componentes permiten controles adicionales para evitar que las cookies se desvíen de su emisor original.

## Ejercicios guiados

1. ¿Cómo se puede leer el contenido del campo `comment`, enviado dentro de una cadena de consulta del método HTTP `GET`, en una función de controlador?

2. Escriba una ruta que utilice el método HTTP `GET` y la ruta `/agent` para enviar de vuelta al cliente el contenido del encabezado `user-agent`.

3. Express.js tiene una característica llamada *route parameters*, donde una ruta como `/user/:name` se puede usar para recibir el parámetro `name` enviado por el cliente. ¿Cómo se puede acceder al parámetro `name` dentro de la función de controlador de la ruta?

## Ejercicios de exploración

1. Si el nombre de host de un servidor es `myserver`, ¿qué ruta Express recibiría el envío en el siguiente formulario?

```
<form action="/contact/feedback" method="post"> ... </form>
```

2. Durante el desarrollo del servidor, el programador no puede leer la propiedad `req.body`, incluso después de verificar que el cliente está enviando correctamente el contenido a través del método HTTP `POST`. ¿Cuál es la causa probable de este problema?

3. ¿Qué sucede cuando el servidor tiene una ruta establecida en `/user/:name` y el cliente realiza una solicitud a `/user/`?

## Resumen

Esta lección explica cómo escribir scripts Express para recibir y manejar solicitudes HTTP. Express utiliza el concepto de *routes* para definir los recursos disponibles para los clientes, lo que le brinda una gran flexibilidad para construir servidores para cualquier tipo de aplicación web. Esta lección abarca los siguientes conceptos y procedimientos:

- Rutas que utilizan los métodos HTTP `GET` y HTTP `POST`.
- Cómo se almacenan los datos del formulario en el objeto `request`.
- Cómo utilizar los parámetros de ruta.
- Personalización de encabezados de respuesta.
- Gestión básica de cookies.

## Respuestas a los ejercicios guiados

1. ¿Cómo se puede leer el contenido del campo `comment`, enviado dentro de una cadena de consulta del método HTTP `GET`, en una función de controlador?

   El campo `comment` está disponible en la propiedad `req.query.comment`.

2. Escriba una ruta que utilice el método HTTP `GET` y la ruta `/agent` para devolver al cliente el contenido del encabezado `user-agent`.

```
app.get('/agent', (req, res) => {
  res.send(req.headers.user-agent)
})
```

3. Express.js tiene una característica llamada *route parameters*, donde una ruta como `/user/:name` se puede usar para recibir el parámetro `name` enviado por el cliente. ¿Cómo se puede acceder al parámetro `name` dentro de la función de controlador de la ruta?

   El parámetro `name` es accesible en la propiedad `req.params.name`.

## Respuestas a los ejercicios de exploración

1. Si el nombre de host de un servidor es `myserver`, ¿qué ruta Express recibiría el envío en el siguiente formulario?

```
<form action="/contact/feedback" method="post"> ... </form>
```

```
app.post('/contact/feedback', (req, res) => {
  ...
})
```

2. Durante el desarrollo del servidor, el programador no puede leer la propiedad `req.body`, incluso después de verificar que el cliente está enviando correctamente el contenido a través del método HTTP `POST`. ¿Cuál es la causa probable de este problema?

   El programador no incluyó el módulo `express.urlencoded`, que permite a Express extraer el cuerpo de una solicitud.

3. ¿Qué sucede cuando el servidor tiene una ruta establecida en `/user/:name` y el cliente realiza una solicitud a `/user/`?

   El servidor emitirá una respuesta `404 Not Found`, porque la ruta requiere que el cliente proporcione el parámetro `:name`.




# 035.2 Lección 2

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 035 Programación NodeJS server |
| **Objetivo:** | 035.2 Conceptos básicos de NodeJS Express |
| **Lección:** | 2 de 2 |

## Introducción

Los servidores web tienen mecanismos muy versátiles para producir respuestas a las solicitudes de los clientes. Para algunas solicitudes, es suficiente que el servidor web proporcione una respuesta estática y sin procesar, porque el recurso solicitado es el mismo para cualquier cliente. Por ejemplo, cuando un cliente solicita una imagen que sea accesible para todos, es suficiente que el servidor envíe el archivo que contiene la imagen.

Pero cuando las respuestas se generan dinámicamente, es posible que deban estar mejor estructuradas que las líneas simples escritas en el script del servidor. En tales casos, es conveniente que el servidor web pueda generar un documento completo, que puede ser interpretado y renderizado por el cliente. En el contexto del desarrollo de aplicaciones web, los documentos HTML se crean comúnmente como plantillas y se mantienen separados del script del servidor, que inserta datos dinámicos en lugares predeterminados en la plantilla adecuada y luego envía la respuesta formateada al cliente.

Las aplicaciones web suelen consumir recursos tanto estáticos como dinámicos. Un documento HTML, incluso si se generó dinámicamente, puede tener referencias a recursos estáticos como archivos e imágenes CSS. Para demostrar cómo Express ayuda a manejar este tipo de demanda,

primero configuraremos un servidor de ejemplo que entrega archivos estáticos y luego implementaremos rutas que generen respuestas estructuradas basadas en plantillas.

## Archivos estáticos

El primer paso es crear el archivo JavaScript que se ejecutará como servidor. Sigamos el mismo patrón cubierto en lecciones anteriores para crear una aplicación Express simple: primero cree un directorio llamado `server` y luego instale los componentes base con el comando `npm`:

```
$ mkdir server
$ cd server/
$ npm init
$ npm install express
```

Para el punto de entrada, se puede usar cualquier nombre de archivo, pero aquí usaremos el nombre de archivo predeterminado: `index.js`. La siguiente lista muestra un archivo básico `index.js` que se utilizará como punto de partida para nuestro servidor:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

No tiene que escribir código explícito para enviar un archivo estático. Express tiene middleware para este propósito, llamado `express.static`. Si su servidor necesita enviar archivos estáticos al cliente, simplemente cargue el middleware `express.static` al comienzo del script:

```
app.use(express.static('public'))
```

El parámetro `public` indica el directorio que almacena los archivos que el cliente puede solicitar. Las rutas solicitadas por los clientes no deben incluir el directorio `public`, sino solo el nombre del archivo o la ruta al archivo relativa al directorio `public`. Para solicitar el archivo `public/layout.css`, por ejemplo, el cliente realiza una solicitud a `/layout.css`.

## Salida formateada

Si bien enviar contenido estático es sencillo, el contenido generado dinámicamente puede variar ampliamente. La creación de respuestas dinámicas con mensajes cortos facilita la prueba de aplicaciones en sus etapas iniciales de desarrollo. Por ejemplo, la siguiente es una ruta de prueba que simplemente envía al cliente un mensaje que envió mediante el método HTTP `POST`. La respuesta puede simplemente replicar el contenido del mensaje en texto sin formato, sin ningún formato:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

Una ruta como esta es un buen ejemplo para usar cuando se aprende Express y para propósitos de diagnóstico, donde una respuesta sin procesar enviada con `res.send()` es suficiente. Pero un servidor útil debe poder producir respuestas más complejas. Continuaremos ahora para desarrollar ese tipo de ruta más sofisticado.

Nuestra nueva aplicación, en lugar de simplemente devolver el contenido de la solicitud actual, mantiene una lista completa de los mensajes enviados en solicitudes anteriores por cada cliente y devuelve la lista de cada cliente cuando se solicita. Una respuesta que combine todos los mensajes es una opción, pero otros modos de salida formateados son más apropiados, especialmente a medida que las respuestas se vuelven más elaboradas.

Para recibir y almacenar los mensajes del cliente enviados durante la sesión actual, primero debemos incluir módulos adicionales para manejar las cookies y los datos enviados a través del método HTTP `POST`. El único propósito del siguiente servidor de ejemplo es registrar los mensajes enviados a través de `POST` y mostrar los mensajes enviados anteriormente cuando el cliente emite una solicitud `GET`. Así que hay dos rutas para la ruta `/`. La primera ruta cumple con las solicitudes realizadas con el método HTTP `POST` y la segunda cumple con las solicitudes realizadas con el método HTTP `GET`:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.static('public'))

const cookieParser = require('cookie-parser')
app.use(cookieParser())

const { v4: uuidv4 } = require('uuid')
```

```
app.use(express.urlencoded({ extended: true }))

// Array to store messages
let messages = []

app.post('/', (req, res) => {

  // Only JSON enabled requests
  if ( req.headers.accept != "application/json" )
  {
    res.sendStatus(404)
    return
  }

  // Locate cookie in the request
  let uuid = req.cookies.uuid

  // If there is no uuid cookie, create a new one
  if ( uuid === undefined )
    uuid = uuidv4()

  // Add message first in the messages array
  messages.unshift({uuid: uuid, message: req.body.message})

  // Collect all previous messages for uuid
  let user_entries = []
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })

  // Update cookie expiration date
  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  // Send back JSON response
  res.json(user_entries)

})

app.get('/', (req, res) => {

  // Only JSON enabled requests
```

```
  if ( req.headers.accept != "application/json" )
  {
    res.sendStatus(404)
    return
  }

  // Locate cookie in the request
  let uuid = req.cookies.uuid

  // Client's own messages
  let user_entries = []

  // If there is no uuid cookie, create a new one
  if ( uuid === undefined ){
    uuid = uuidv4()
  }
  else {
    // Collect messages for uuid
    messages.forEach( (entry) => {
      if ( entry.uuid == req.cookies.uuid )
        user_entries.push(entry.message)
    })
  }

  // Update cookie expiration date
  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  // Send back JSON response
  res.json(user_entries)

})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Mantuvimos la configuración de archivos estáticos en la parte superior, porque pronto será útil proporcionar archivos estáticos como `layout.css`. Además del middleware `cookie-parser` presentado en el capítulo anterior, el ejemplo también incluye el middleware `uuid` para generar un número de identificación único que se pasa como una cookie a cada cliente que envía un mensaje. Si aún no está instalado en el directorio del servidor de ejemplo, estos módulos se pueden instalar con el comando `npm install cookie-parser uuid`.

El arreglo global llamado `messages` almacena los mensajes enviados por todos los clientes. Cada elemento de este arreglo consta de un objeto con las propiedades `uuid` y `message`.

Lo que es realmente nuevo en este script es el método `res.json()`, usado al final de las dos rutas para generar una respuesta en formato JSON con el arreglo que contiene los mensajes ya enviados por el cliente:

```
// Send back JSON response
res.json(user_entries)
```

JSON es un formato de texto sin formato que le permite agrupar un conjunto de datos en una sola estructura que es asociativa: es decir, el contenido se expresa como claves y valores. JSON es particularmente útil cuando el cliente va a procesar las respuestas. Con este formato, un objeto o arreglo de JavaScript se puede reconstruir fácilmente en el lado del cliente con todas las propiedades e índices del objeto original en el servidor.

Debido a que estamos estructurando cada mensaje en JSON, rechazamos las solicitudes que no contienen `application/json` en su encabezado `accept`:

```
// Only JSON enabled requests
if ( req.headers.accept != "application/json" )
{
  res.sendStatus(404)
  return
}
```

No se aceptará una solicitud realizada con un comando simple `curl` para insertar un nuevo mensaje, porque `curl` por defecto no especifica `application/json` en el encabezado `accept`:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt
Not Found
```

La opción `-H "accept: application/json"` cambia el encabezado de la solicitud para especificar el formato de la respuesta, que esta vez será aceptado y respondido en el formato especificado:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt -H "accept: application/json"
["My first message"]
```

Obtener mensajes usando la otra ruta se hace de una manera similar, pero esta vez usando el método HTTP `GET`:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -H "accept:
application/json"
["Another message","My first message"]
```

## Plantillas

Las respuestas en formatos como JSON son convenientes para la comunicación entre programas, pero el objetivo principal de la mayoría de los servidores de aplicaciones web es producir contenido HTML para consumo humano. Incrustar código HTML dentro del código JavaScript no es una buena idea, porque mezclar lenguajes en el mismo archivo hace que el programa sea más susceptible a errores y perjudica el mantenimiento del código.

Express puede trabajar con diferentes *motores de plantillas* que separan el HTML para contenido dinámico; la lista completa se puede encontrar en el Sitio de motores de plantillas express. Uno de los motores de plantillas más populares es *Embedded JavaScript* (EJS), que le permite crear archivos HTML con etiquetas específicas para la inserción de contenido dinámico.

Al igual que otros componentes Express, EJS debe instalarse en el directorio donde se ejecuta el servidor:

```
$ npm install ejs
```

A continuación, el motor EJS debe configurarse como el renderizador predeterminado en el script del servidor (cerca del comienzo del archivo `index.js`, antes de las definiciones de ruta):

```
app.set('view engine', 'ejs')
```

La respuesta generada con la plantilla se envía al cliente con la función `res.render()`, que recibe como parámetros el nombre del archivo de la plantilla y un objeto que contiene valores que serán accesibles desde dentro de la plantilla. Las rutas utilizadas en el ejemplo anterior se pueden reescribir para generar respuestas HTML y JSON:

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid

  if ( uuid === undefined )
    uuid = uuidv4()

  messages.unshift({uuid: uuid, message: req.body.message})

  let user_entries = []
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })

  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  if ( req.headers.accept == "application/json" )
    res.json(user_entries)
  else
    res.render('index', {title: "My messages", messages: user_entries})

})

app.get('/', (req, res) => {

  let uuid = req.cookies.uuid

  let user_entries = []

  if ( uuid === undefined ){
    uuid = uuidv4()
  }
  else {
    messages.forEach( (entry) => {
      if ( entry.uuid == req.cookies.uuid )
        user_entries.push(entry.message)
    })
  }

  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
```

```
  res.cookie('uuid', uuid, { expires: expires })

  if ( req.headers.accept == "application/json" )
    res.json(user_entries)
  else
    res.render('index', {title: "My messages", messages: user_entries})

})
```

Tenga en cuenta que el formato de la respuesta depende del encabezado `accept` que se encuentra en la solicitud:

```
if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})
```

Se envía una respuesta en formato JSON solo si el cliente la solicita explícitamente. De lo contrario, la respuesta se genera a partir de la plantilla `index`. El mismo arreglo `user_entries` alimenta tanto la salida JSON como la plantilla, pero el objeto utilizado como parámetro para esta última también tiene la propiedad `title: "My messages"`, que se utilizará como título dentro de la plantilla.

## Plantillas HTML

Al igual que los archivos estáticos, los archivos que contienen plantillas HTML residen en su propio directorio. De forma predeterminada, EJS asume que los archivos de plantilla están en el directorio `views/`. En el ejemplo, se utilizó una plantilla llamada `index`, por lo que EJS busca el archivo `views/index.ejs`. La siguiente lista es el contenido de una plantilla simple `views/index.ejs` que se puede usar con el código de ejemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title><%= title %></title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
</p>
</form>

<ul>
<% messages.forEach( (message) => { %>
<li><%= message %></li>
<% }) %>
</ul>

</div>

</body>
</html>
```

La primera etiqueta EJS especial es el elemento `<title>` en la sección `<head>`:

```
<%= title %>
```

Durante el proceso de renderizado, esta etiqueta especial será reemplazada por el valor de la propiedad `title` del objeto pasado como parámetro a la función `res.render()`.

La mayor parte de la plantilla se compone de código HTML convencional, por lo que la plantilla contiene el formulario HTML para enviar nuevos mensajes. El servidor de prueba responde a los métodos HTTP `GET` y `POST` para la misma ruta `/`, de ahí los atributos `action="/"` y `method="post"` en la etiqueta del formulario.

Otras partes de la plantilla son una mezcla de código HTML y etiquetas EJS. EJS tiene etiquetas para propósitos específicos dentro de la plantilla:

**`<% … %>`**

Inserta el control de flujo. Esta etiqueta no inserta contenido directamente, pero se puede usar con estructuras de JavaScript para elegir, repetir o suprimir secciones de HTML. Ejemplo iniciando un bucle: `<% messages.forEach( (message) => { %>`

**`<%# … %>`**

Define un comentario, cuyo contenido es ignorado por el analizador. A diferencia de los comentarios escritos en HTML, estos comentarios no son visibles para el cliente.

**`<%= … %>`**

Inserta el contenido de escape de la variable. Es importante escapar del contenido desconocido para evitar la ejecución de código JavaScript, que puede abrir lagunas para los ataques de Scripting entre sitios (XSS). Ejemplo: `<%= title %>`

**`<%- … %>`**

Inserta el contenido de la variable sin escapar.

La combinación de código HTML y etiquetas EJS es evidente en el fragmento donde los mensajes del cliente se representan como una lista HTML:

```
<ul>
<% messages.forEach( (message) => { %>
<li><%= message %></li>
<% }) %>
</ul>
```

En este fragmento, la primera etiqueta `<% … %>` inicia una declaración `forEach` que recorre todos los elementos del arreglo `message`. Los delimitadores `<%` y `%>` le permiten controlar los fragmentos de HTML. Se producirá un nuevo elemento de lista HTML, `<li><%= message %></li>`, para cada elemento de `messages`. Con estos cambios, el servidor enviará la respuesta en HTML cuando se reciba una solicitud como la siguiente:

```
$ curl http://myserver:8080/ --data message="This time" -c cookies.txt -b
cookies.txt
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My messages</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
</p>
</form>

<ul>

<li>This time</li>

<li>in HTML</li>

</ul>

</div>

</body>
</html>
```

La separación entre el código para procesar las solicitudes y el código para presentar la respuesta hace que el código sea más limpio y permite que un equipo divida el desarrollo de aplicaciones entre personas con distintas especialidades. Un diseñador web, por ejemplo, puede centrarse en los archivos de plantilla en `views/` y hojas de estilo relacionadas, que se proporcionan como archivos estáticos almacenados en el directorio `public/` del servidor de ejemplo.

# Ejercicios guiados

1. ¿Cómo debería configurarse `express.static` para que los clientes puedan solicitar archivos en el directorio `assets`?

2. ¿Cómo se puede identificar el tipo de respuesta, que se especifica en el encabezado de la solicitud, dentro de una ruta Express?

3. ¿Qué método del parámetro de ruta `res` (respuesta) genera una respuesta en formato JSON a partir de un arreglo de JavaScript llamado `content`?

## Ejercicios de exploración

1. Por defecto, los archivos de plantilla Express están en el directorio `views`. ¿Cómo se puede modificar esta configuración para que los archivos de plantilla se almacenen en `templates`?

2. Supongamos que un cliente recibe una respuesta HTML sin título (es decir, `<title> </title>`). Después de verificar la plantilla EJS, el desarrollador encuentra la etiqueta `<title><% title %></title>` en la sección `head` del archivo. ¿Cuál es la causa probable del problema?

3. Utilice etiquetas de plantilla EJS para escribir una etiqueta HTML `<h2></h2>` con el contenido de la variable JavaScript `h2`. Esta etiqueta debe ser renderizada solo si la variable `h2` no está vacía.

## Resumen

Esta lección cubre los métodos básicos que proporciona Express.js para generar respuestas estáticas y formateadas pero dinámicas. Se requiere poco esfuerzo para configurar un servidor HTTP para archivos estáticos y el sistema de plantillas EJS proporciona una manera fácil de generar contenido dinámico a partir de archivos HTML. Esta lección abarca los siguientes conceptos y procedimientos:

- Uso de `express.static` para respuestas de archivos estáticos.
- Cómo crear una respuesta para que coincida con el campo de tipo de contenido en el encabezado de la solicitud.
- Respuestas estructuradas en JSON.
- Uso de etiquetas EJS en plantillas basadas en HTML.

## Respuestas a los ejercicios guiados

1. ¿Cómo debería configurarse `express.static` para que los clientes puedan solicitar archivos en el directorio `assets`?

   Se debe agregar una llamada a `app.use(express.static('assets'))` al script del servidor.

2. ¿Cómo se puede identificar el tipo de respuesta, que se especifica en el encabezado de la solicitud, dentro de una ruta Express?

   El cliente establece tipos aceptables en el campo de encabezado `accept`, que se asigna a la propiedad `req.headers.accept`.

3. ¿Qué método del parámetro de ruta `res` (respuesta) genera una respuesta en formato JSON a partir de un arreglo de JavaScript llamado `content`?

   El método `res.json()`: `res.json(content)`.

## Respuestas a los ejercicios de exploración

1. Por defecto, los archivos de plantilla Express están en el directorio `views`. ¿Cómo se puede modificar esta configuración para que los archivos de plantilla se almacenen en `templates`?

   El directorio se puede definir en la configuración inicial del script con `app.set('views', './templates')`.

2. Supongamos que un cliente recibe una respuesta HTML sin título (es decir, `<title> </title>`). Después de verificar la plantilla EJS, el desarrollador encuentra la etiqueta `<title><% title %></title>` en la sección `head` del archivo. ¿Cuál es la causa probable del problema?

   La etiqueta `<%= %>` debe usarse para encerrar el contenido de una variable, como en `<%= title %>`.

3. Use etiquetas de plantilla EJS para escribir una etiqueta HTML `<h2></h2>` con el contenido de la variable JavaScript `h2`. Esta etiqueta debe representarse solo si la variable `h2` no está vacía.

   ```
   <% if ( h2 != "" ) { %>
   <h2><%= h2 %></h2>
   <% } %>
   ```



# 035.3 Conceptos básicos de SQL

**Referencia al objetivo del LPI**

Web Development Essentials version 1.0, Exam 030, Objective 035.3

**Peso**

3

**Áreas de conocimiento clave**

- Establecer una conexión de base de datos desde NodeJS
- Recuperar datos de la base de datos en NodeJS
- Ejecutar consultas SQL desde NodeJS
- Cree consultas SQL simples que excluyan las uniones
- Comprender las claves primarias
- Variables de escape utilizadas en consultas SQL
- Conocimiento de las inyecciones de SQL

**Lista parcial de archivos, términos y utilidades**

- Módulo NPM `sqlite3`
- `Database.run()`, `Database.close()`, `Database.all()`, `Database.get()`, `Database.each()`
- `CREATE TABLE`
- `INSERT`, `SELECT`, `DELETE`, `UPDATE`




# 035.3 Lección 1

| | |
|---|---|
| **Certificación:** | Conceptos básicos de desarrollo web |
| **Versión:** | 1.0 |
| **Tema:** | 035 Programación NodeJS server |
| **Objetivo:** | 035.3 Conceptos básicos de SQL |
| **Lección:** | 1 de 1 |

## Introducción

Aunque puede escribir sus propias funciones para implementar el almacenamiento persistente, puede ser más conveniente utilizar un sistema de administración de bases de datos para acelerar el desarrollo y garantizar una mejor seguridad y estabilidad para los datos con formato de tabla. La estrategia más popular para almacenar datos organizados en tablas interrelacionadas, especialmente cuando esas tablas son muy consultadas y actualizadas, es instalar una base de datos relacional que admita *Structured Query Language* (SQL), un lenguaje orientado a bases de datos relacionales. Node.js es compatible con varios sistemas de administración de bases de datos SQL. Siguiendo los principios de portabilidad y ejecución del espacio de usuario adoptados por Node.js Express, SQLite es una opción adecuada para el almacenamiento persistente de datos utilizados por este tipo de servidor HTTP.

### SQL

El lenguaje de consulta estructurado es específico de las bases de datos. Las operaciones de escritura y lectura se expresan en instrucciones llamadas *sentencias* y *consultas*. Tanto las sentencias como las consultas se componen de *cláusulas*, que definen las condiciones para ejecutar la operación.

Los nombres y las direcciones de correo electrónico, por ejemplo, se pueden almacenar en una tabla de base de datos que contiene los campos `name` e `email`. Una base de datos puede contener varias tablas, por lo que cada tabla debe tener un nombre único. Si usamos el nombre `contacts` para la tabla de nombres y correos electrónicos, se puede insertar un nuevo registro con la siguiente *declaración*:

```
INSERT INTO contacts (name, email) VALUES ("Carol", "carol@example.com");
```

Esta declaración de inserción se compone de la cláusula `INSERT INTO`, que define la tabla y los campos donde se insertarán los datos. La segunda cláusula, `VALUES`, establece los valores que se insertarán. No es necesario usar mayúsculas en las cláusulas, pero es una práctica común para reconocer mejor las palabras claves SQL dentro de una declaración o consulta.

Una consulta en la tabla de contactos se realiza de manera similar, pero usando la cláusula `SELECT`:

```
SELECT email FROM contacts;
dave@example.com
carol@example.com
```

En este caso, la cláusula `SELECT email` selecciona un campo de las entradas en la tabla de `contacts`. La cláusula `WHERE` restringe la consulta a filas específicas:

```
SELECT email FROM contacts WHERE name = "Dave";
dave@example.com
```

SQL tiene muchas otras cláusulas, y veremos algunas de ellas en secciones posteriores. Pero primero es necesario ver cómo integrar la base de datos SQL con Node.js.

## SQLite

SQLite es probablemente la solución más simple para incorporar características de base de datos SQL en una aplicación. A diferencia de otros sistemas de administración de bases de datos populares, SQLite no es un servidor de base de datos al que se conecta un cliente. En cambio, SQLite proporciona un conjunto de funciones que permiten al desarrollador crear una base de datos como un archivo convencional. En el caso de un servidor HTTP implementado con Node.js Express, este archivo generalmente se encuentra en el mismo directorio que el script del servidor.

Antes de usar SQLite en Node.js, necesita instalar el módulo `sqlite3`. Ejecute el siguiente comando en el directorio de instalación del servidor; es decir, el directorio que contiene el script Node.js que ejecutará.

```
$ npm install sqlite3
```

Tenga en cuenta que hay varios módulos que admiten SQLite, como `better-sqlite3`, cuyo uso es sutilmente diferente de `sqlite3`. Los ejemplos de esta lección son para el módulo `sqlite3`, por lo que es posible que no funcionen como se esperaba si elige otro módulo.

## Abrir la base de datos

Para demostrar cómo un servidor Node.js Express puede funcionar con una base de datos SQL, escribamos un script que almacene y muestre los mensajes enviados por un cliente identificado por una cookie. El cliente envía los mensajes a través del método HTTP POST y la respuesta del servidor puede formatearse como JSON o HTML (a partir de una plantilla), según el formato solicitado por el cliente. Esta lección no entrará en detalles sobre el uso de métodos, cookies y plantillas HTTP. Los fragmentos de código que se muestran aquí asumen que ya tiene un servidor Node.js Express donde estas funciones están configuradas y disponibles.

La forma más sencilla de almacenar los mensajes enviados por el cliente es almacenarlos en un arreglo global, donde cada mensaje enviado previamente se asocia con una clave de identificación única para cada cliente. Esta clave se puede enviar al cliente como una cookie, que se presenta al servidor en futuras solicitudes para recuperar sus mensajes anteriores.

Sin embargo, este enfoque tiene una debilidad: debido a que los mensajes se almacenan solo en un arreglo global, todos los mensajes se perderán cuando finalice la sesión actual del servidor. Ésta es una de las ventajas de trabajar con bases de datos, porque los datos se almacenan de forma persistente y no se pierden si se reinicia el servidor.

Usando el archivo `index.js` como el script principal del servidor, podemos incorporar el módulo `sqlite3` e indicar el archivo que sirve como base de datos, de la siguiente manera:

```
const sqlite3 = require('sqlite3')
const db = new sqlite3.Database('messages.sqlite3');
```

Si aún no existe, el archivo `messages.sqlite3` se creará en el mismo directorio que el archivo `index.js`. Dentro de este único archivo, se almacenarán todas las estructuras y los datos respectivos. Todas las operaciones de la base de datos realizadas en el script serán intermediadas por la constante `db`, que es el nombre dado al nuevo objeto `sqlite3` que abre el archivo `messages.sqlite3`.

## Estructura de una Tabla

No se pueden insertar datos en la base de datos hasta que se cree al menos una tabla. Las tablas se crean con la instrucción `CREATE TABLE`:

```
db.run('CREATE TABLE IF NOT EXISTS messages (id INTEGER PRIMARY KEY AUTOINCREMENT,
uuid CHAR(36), message TEXT)')
```

El método `db.run()` se utiliza para ejecutar sentencias SQL en la base de datos. La declaración en sí está escrita como un parámetro para el método. Aunque las instrucciones SQL deben terminar con un punto y coma cuando se ingresan en un procesador de línea de comandos, el punto y coma es opcional en las instrucciones que se pasan como parámetros en un programa.

Debido a que el método `run` se ejecutará cada vez que se ejecute el script con `node index.js`, la declaración SQL incluye la cláusula condicional `IF NOT EXISTS` para evitar errores en ejecuciones futuras cuando la tabla `messages` ya exista.

Los campos que componen la tabla `messages` son `id`, `uuid` y `message`. El campo `id` es un número entero único que se utiliza para identificar cada entrada en la tabla, por lo que se crea como `PRIMARY KEY`. Las claves primarias no pueden ser nulas y no puede haber dos claves primarias idénticas en la misma tabla. Por lo tanto, casi todas las tablas SQL tienen una clave principal para rastrear el contenido de la tabla. Aunque es posible elegir explícitamente el valor de la clave primaria de un nuevo registro (siempre que aún no exista en la tabla), es conveniente que la clave se genere automáticamente. La marca `AUTOINCREMENT` en el campo `id` se utiliza para este propósito.

**NOTE**

La configuración explícita de claves primarias en SQLite es opcional, porque SQLite mismo crea una clave primaria automáticamente. Como se indica en la documentación de SQLite: "En SQLite, las filas de la tabla normalmente tienen un entero de 64 bits con signo `ROWID` que es único entre todas las filas de la misma tabla. Si una tabla contiene una columna de tipo `INTEGER PRIMARY KEY`, luego, esa columna se convierte en un alias para el `ROWID`. Luego, puede acceder al `ROWID` utilizando cualquiera de los cuatro nombres diferentes, los tres nombres originales descritos anteriormente o el nombre dado a la columna `INTEGER PRIMARY KEY`. Todos estos nombres son alias entre sí y funcionan igual de bien en cualquier contexto."

Los campos `uuid` y `message` almacenan la identificación del cliente y el contenido del mensaje, respectivamente. Un campo de tipo `CHAR (36)` almacena una cantidad fija de 36 caracteres, y un campo de tipo `TEXT` almacena textos de longitud arbitraria.

## Entrada de datos

La función principal de nuestro servidor de ejemplo es almacenar mensajes que están vinculados al cliente que los envió. El cliente envía el mensaje en el campo `message` en el cuerpo de la solicitud enviada con el método HTTP POST. La identificación del cliente está en una cookie llamada `uuid`. Con esta información, podemos escribir la ruta Express para insertar nuevos mensajes en la base de datos:

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid

  if ( uuid === undefined )
    uuid = uuidv4()

  // Insert new message into the database
  db.run('INSERT INTO messages (uuid, message) VALUES (?, ?)', uuid, req.body
.message)

  // If an error occurrs, err object contains the error message.
  db.all('SELECT id, message FROM messages WHERE uuid = ?', uuid, (err, rows) => {

    let expires = new Date(Date.now());
    expires.setDate(expires.getDate() + 30);
    res.cookie('uuid', uuid, { expires: expires })

    if ( req.headers.accept == "application/json" )
      res.json(rows)
    else
      res.render('index', {title: "My messages", rows: rows})

  })

})
```

Esta vez, el método `db.run()` ejecuta una instrucción de inserción, pero tenga en cuenta que el `uuid` y `req.body.message` no se escriben directamente en la línea de instrucción. En cambio, se sustituyeron los valores por signos de interrogación. Cada signo de interrogación corresponde a un parámetro que sigue a la declaración SQL en el método `db.run()`.

El uso de signos de interrogación como marcadores de posición en la declaración que se ejecuta en la base de datos facilita a SQLite distinguir entre los elementos estáticos de la declaración y sus datos variables. Esta estrategia permite que SQLite *escape* o *sanitice* el contenido de las variables que son

parte de la declaración, evitando una brecha de seguridad común llamada *SQL injection*. En ese ataque, los usuarios malintencionados insertan declaraciones SQL en los datos variables con la esperanza de que las declaraciones se ejecuten inadvertidamente; sanitizar frustra el ataque al deshabilitar los caracteres peligrosos en los datos.

## Consultas

Como se muestra en el código de ejemplo, nuestra intención es usar la misma ruta para insertar nuevos mensajes en la base de datos y generar la lista de mensajes enviados previamente. El método `db.all()` devuelve la colección de todas las entradas en la tabla que coinciden con los criterios definidos en la consulta.

A diferencia de las sentencias realizadas por `db.run()`, `db.all()` genera una lista de registros que son manejados por la función de flecha designada en el último parámetro:

```
(err, rows) => {}
```

Esta función, a su vez, toma dos parámetros: `err` y `rows`. El parámetro `err` se utilizará si se produce un error que impida la ejecución de la consulta. Si tiene éxito, todos los registros están disponibles en el arreglo de `rows`, donde cada elemento es un objeto correspondiente a un solo registro de la tabla. Las propiedades de este objeto corresponden a los nombres de campo indicados en la consulta: `uuid` y `message`.

El arreglo de `rows` es una estructura de datos de JavaScript. Como tal, se puede utilizar para generar respuestas con métodos proporcionados por Express, como `res.json()` y `res.render()`. Cuando se representa dentro de una plantilla EJS, un bucle convencional puede enumerar todos los registros:

```
<ul>
<% rows.forEach( (row) => { %>
<li><strong><%= row.id %></strong>: <%= row.message %></li>
<% }) %>
</ul>
```

En lugar de llenar el arreglo de `rows` con todos los registros devueltos por la consulta, en algunos casos puede ser más conveniente tratar cada registro individualmente con el método `db.each()`. La sintaxis del método `db.each()` es similar al método `db.all()`, pero el parámetro `row` en `(err, row) ⇒ {}` coincide con un solo registro a la vez.

## Modificar el contenido de la base de datos

Hasta ahora, nuestro cliente solo puede agregar y consultar mensajes en el servidor. Dado que el cliente ahora conoce el `id` de los mensajes enviados anteriormente, podemos implementar una función para modificar un registro específico. El mensaje modificado también se puede enviar a una ruta del método HTTP POST, pero esta vez con un parámetro de ruta para capturar el `id` dado por el cliente en la ruta de la solicitud:

```
app.post('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if ( uuid === undefined ){
    uuid = uuidv4()
    // 401 Unauthorized
    res.sendStatus(401)
  }
  else {

    // Update the stored message
    // using named parameters
    let param = {
      $message: req.body.message,
      $id: req.params.id,
      $uuid: uuid
    }

    db.run('UPDATE messages SET message = $message WHERE id = $id AND uuid = $uuid
', param, function(err){

      if ( this.changes > 0 )
      {
        // A 204 (No Content) status code means the action has
        // been enacted and no further information is to be supplied.
        res.sendStatus(204)
      }
      else
        res.sendStatus(404)

    })
  }
})
```

Esta ruta demuestra cómo usar las cláusulas `UPDATE` y `WHERE` para modificar un registro existente.

Una diferencia importante con los ejemplos anteriores es el uso de *parámetros con nombre*, donde los valores se agrupan en un solo objeto (`param`) y se pasan al método `db.run()` en lugar de especificar cada valor por sí mismo. En este caso, los nombres de los campos (precedidos por `$`) son las propiedades del objeto. Los parámetros con nombre permiten el uso de nombres de campo (precedidos por `$`) como marcadores de posición en lugar de signos de interrogación.

Una declaración como la del ejemplo no causará ninguna modificación a la base de datos si la condición impuesta por la cláusula `WHERE` no coincide con algún registro en la tabla. Para evaluar si la declaración modificó algún registro, se puede usar una función de devolución de llamada como último parámetro del método `db.run()`. Dentro de la función, el número de registros modificados se puede consultar desde `this.changes`. Tenga en cuenta que las funciones de flecha no se pueden usar en este caso, porque solo las funciones regulares de la forma `function(){}` definen el objeto `this`.

Eliminar un registro es muy similar a modificarlo. Podemos, por ejemplo, continuar usando el parámetro de ruta `:id` para identificar el mensaje a eliminar, pero esta vez en una ruta invocada por el método HTTP DELETE del cliente:

```
app.delete('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if ( uuid === undefined ){
    uuid = uuidv4()
    res.sendStatus(401)
  }
  else {
    // Named parameters
    let param = {
    $id: req.params.id,
    $uuid: uuid
    }

    db.run('DELETE FROM messages WHERE id = $id AND uuid = $uuid', param, function
(err){
      if ( this.changes > 0 )
        res.sendStatus(204)
      else
        res.sendStatus(404)
    })
  }
})
```

Los registros se eliminan de una tabla con la cláusula `DELETE FROM`. De nuevo usamos la función de

devolución de llamada para evaluar cuántas entradas se han eliminado de la tabla.

## Cerrar la base de datos

Una vez definido, se puede hacer referencia al objeto `db` en cualquier momento durante la ejecución del script, porque el archivo de la base de datos permanece abierto durante la sesión actual. No es común cerrar la base de datos mientras se ejecuta el script.

Sin embargo, una función para cerrar la base de datos es útil para evitar cerrar abruptamente la base de datos cuando finaliza el proceso del servidor. Aunque es poco probable, el cierre abrupto de la base de datos puede generar incoherencias si los datos en memoria aún no se han confirmado en el archivo. Por ejemplo, un cierre abrupto de la base de datos con pérdida de datos puede ocurrir si el usuario finaliza el script presionando el atajo de teclado Ctrl+C.

En el escenario Ctrl+C que se acaba de describir, el método `process.on()` puede interceptar las señales enviadas por el sistema operativo y ejecutar un apagado ordenado tanto de la base de datos como del servidor:

```
process.on('SIGINT', () => {
  db.close()
  server.close()
  console.log('HTTP server closed')
})
```

El atajo Ctrl+C invoca la señal del sistema operativo `SIGINT`, que termina un programa en primer plano en la terminal. Antes de finalizar el proceso al recibir la señal `SIGINT`, el sistema invoca la función de devolución de llamada (el último parámetro en el método `process.on()`). Dentro de la función de devolución de llamada, puede poner cualquier código de limpieza, en particular el método `db.close()` para cerrar la base de datos y `server.close()`, que cierra elegantemente la propia instancia Express.

## Ejercicios guiados

1. ¿Cuál es el propósito de una llave primaria en una tabla de base de datos SQL?

2. ¿Cuál es la diferencia entre consultar usando `db.all()` y `db.each()`?

3. ¿Por qué es importante utilizar marcadores de posición y no incluir los datos enviados por el cliente directamente en una instrucción o consulta SQL?

## Ejercicios de exploración

1. ¿Qué método del módulo `sqlite3` se puede utilizar para devolver solo una entrada de tabla, incluso si la consulta coincide con varias entradas?

2. Suponga que el arreglo de `rows` se pasó como un parámetro a una función de devolución de llamada y contiene el resultado de una consulta realizada con `db.all()`. ¿Cómo se puede hacer referencia a un campo llamado `price`, que está presente en la primera posición de `rows`, dentro de la función de devolución de llamada?

3. El método `db.run ()` ejecuta sentencias de modificación de la base de datos, como `INSERT INTO`. Después de insertar un nuevo registro en una tabla, ¿cómo podría recuperar la clave principal del registro recién insertado?

## Resumen

Esta lección cubre el uso básico de bases de datos SQL dentro de las aplicaciones Node.js Express. El módulo `sqlite3` ofrece una forma sencilla de almacenar datos persistentes en una base de datos SQLite, donde un solo archivo contiene toda la información y no requiere un servidor de base de datos especializado. Esta lección abarca los siguientes conceptos y procedimientos:

- Cómo establecer una conexión a la base de datos desde Node.js.
- Cómo crear una tabla simple y el rol de las claves primarias.
- Usar la instrucción SQL `INSERT INTO` para agregar nuevos datos desde el script.
- Consultas SQL utilizando métodos estándares SQLite y funciones de devolución de llamada.
- Modificación de datos en la base de datos usando sentencias SQL `UPDATE` y `DELETE`.

## Respuestas a los ejercicios guiados

1. ¿Cuál es el propósito de una llave primaria en una tabla de base de datos SQL?

   La llave primaria es el campo de identificación único para cada registro dentro de una tabla de una base de datos.

2. ¿Cuál es la diferencia entre consultar usando `db.all()` y `db.each()`?

   El método `db.all()` invoca la función de devolución de llamada con un único arreglo que contiene todas las entradas correspondientes a la consulta. El método `db.each()` invoca la función de devolución de llamada para cada fila de resultados.

3. ¿Por qué es importante utilizar marcadores de posición y no incluir los datos enviados por el cliente directamente en una instrucción o consulta SQL?

   Con los marcadores de posición, los datos enviados por el usuario se escapan antes de ser incluidos en la consulta o declaración. Esto dificulta los ataques de inyección de SQL, donde las sentencias de SQL se colocan dentro de datos variables en un intento de realizar operaciones arbitrarias en la base de datos.

## Respuestas a los ejercicios de exploración

1. ¿Qué método del módulo `sqlite3` se puede utilizar para devolver solo una entrada de tabla, incluso si la consulta coincide con varias entradas?

   El método `db.get()` tiene la misma sintaxis que `db.all()`, pero devuelve solo la primera entrada correspondiente a la consulta.

2. Suponga que el arreglo de `rows` se pasó como un parámetro a una función de devolución de llamada y contiene el resultado de una consulta realizada con `db.all()`. ¿Cómo se puede hacer referencia a un campo llamado `price`, que está presente en la primera posición de `rows`, dentro de la función de devolución de llamada?

   Cada elemento en `rows` es un objeto cuyas propiedades corresponden a los nombres de los campos de la base de datos. Por tanto, el valor del campo `price` en el primer resultado está en las `rows[0].price`.

3. El método `db.run ()` ejecuta sentencias de modificación de la base de datos, como `INSERT INTO`. Después de insertar un nuevo registro en una tabla, ¿cómo podría recuperar la clave principal del registro recién insertado?

   Una función regular de la forma `function(){}` se puede utilizar como función de devolución de llamada del método `db.run()`. En su interior, la propiedad `this.lastID` contiene el valor de la clave principal del último registro insertado.

# Pie de imprenta



PDF generado: 2022-04-21



LPI Learning Materials son una iniciativa del Linux Professional Institute (https://lpi.org). Los materiales y sus traducciones pueden encontrarse en https://learning.lpi.org.

Para preguntas y comentarios sobre esta edición, así como sobre todo el proyecto, escriba un correo electrónico a: learning@lpi.org.